SeqAn Manual

Release 2.2.0

The SeqAn Team

Jul 28, 2017

Tutorials

Welcome to the manual pages for the SeqAn library!

SeqAn is a C++ template library for the analysis of biological sequences. As such, it contains algorithms and data structures for

- string representation and their manipluation,
- online and indexed string search,
- efficient I/O of bioinformatics file formats,
- sequence alignments, and
- many, many more.

CHAPTER 1

Requirements

Well, as SeqAn is written in C++ it might not come as a surprise, that you should bring some basic knowledge about the C++ programming language. We made quite an effort to not depend on other libraries, than the on-board tools already bring. This means, to learn SeqAn it suffices in the beginning to only know about C++ and the STL which is the standard template library defined by the ISO C++ committee. The rest will be discussed in the subsequent tutorials step by step.

Before we start, here is a strong advice! If you are diving into C++ for the first time, because you are new to programming or switched from another programming language, then we recommend, you first stroll through the C++ FAQs to acquaint yourself with C++. There you can find many useful tips about C++ and get some further readings. It also will introduce you to the paradigms that we used for designing this library. In the Getting Started section we will introduce you to the design decisions of SeqAn and lay down some of the basic programming paradigms we follow to make this library so efficient. If you never heard about these paradigms, dont't worry. We will give you code examples, which you can try out on your own. Never forgot, there is no better way to learn a new language or language feature, than to actually program with it. So keep your fingers attached to the keyboard and let's start right away!

If you didn't install SeqAn yet, please follow the *User Guide* instructions to install SeqAn first. After that you should continue with the tutorials.

Hint: Please note, that although we try hard to provide a very comprehensive list of topics, it is not always possible to cover every angle of the library and its features. The tutorials are thought as a first place to start. If you are more experienced with SeqAn you can use the *API documentation* in addition to search for specific functions or classes.

CHAPTER 2

Tutorials

The tutorial section is organized such that you can efficiently search for a specific topic you want to learn more about. Each tutorial takes 30 to 60 minutes of your time for learning how to use SeqAn. So buckle up and jump right into using SeqAn using our tutorials!

Getting Started These articles are required for every one that is new to SeqAn. Take your time and study these documents thoroughly, as they describe the fundamental concepts and design decisions of the library. Everything else depends on these informations.

Data Structures In the data structure tutorials we introduce you to the main data structures of this library and their usage. Beginners should start with the *Sequence tutorial*, and then continue with the *Alingment tutorials*. After that beginners should continue with the *Alignment Algorithm tutorials*.

Algorithms In this section we explain several different algorithms that are crucial for many bioinformatics applications. This includes pattern matching, dynamic programming algorithms for sequence alignments, seed extension and many more. Beginners that come from the tutorials about data structures should either continue with *Online Pattern Matching* or with the *DP Alignment Algorithms*.

Input/Output On this page you will learn how to read/write and work with common bioinformatic file formats, such as FASTA, BAM, BED, VCF files, and more. Beginners should start with the *File I/O Overview*. This tutorial introduces you to the basic I/O concepts and data structures.

How-Tos The how-to page is divided into Recipes and Use Cases. The former section gives you some useful hints about miscellaneous topics. The latter section describes how some use cases can be solved with SeqAn. Things presented here are for experienced SeqAn users. If you are a beginner, first have a look at the tutorials above.

Workflows These tutorials teach you how to integrate your application into workflow engines like KNIME or Galaxy.

CHAPTER 3

Infrastructure

User Guide These articles describe how to get SeqAn, how to use it in your application and explain things you need to consider when building. Everyone should read it.

Contributer Guide Anyone who wants to contribute code or documentation to SeqAn should read this. You will learn about the conventions and coding style.

Team Guide These pages cover the structure of the SeqAn repository, the git workflow and explain release procedures. All SeqAn team members should read this; and also downstream package maintainers.

CHAPTER 4

API Documentation

The API documentation can be found here.

Getting Started

These Tutorials will help you with your first steps in SeqAn.

Important: Before you start, make sure that you have installed SeqAn correctly by following the *User Guide*!

ToC

Contents

Background and Motivation

Learning Objective You will learn about the design goals and fundamental ideas used in the SeqAn library. Also, you will see how the SeqAn library can be generic while still retaining high performance.

Difficulty Very basic

Duration Take the time you need!

Prerequisites Basic C or C++ knowledge

Hi, we are glad you made it here. You being here, and reading these lines means you are eager to learn more about SeqAn and this is the right place to start. In this tutorial, we will give you an overview about the design goals, design decisions of the SeqAn library, and explain the motivation for these decisions. The next chapter *First Steps* will flesh out the most important points of this chapter with code examples of everyday SeqAn use.

Library Design Aims

The following lists some library design aims of the SeqAn library. Note that they are contradicting. The focus is on efficiency but small trade-offs are allowed to improve consistency and ease of use.

1. **Efficiency**. The focus of SeqAn is to provide a library of efficient and reusable algorithmic components for biological sequence analysis. Algorithms should have good practical implementations with low overhead, even at the cost of being harder to use.
2. **Consistency**. Be consistent wherever possible, even at slight costs of efficiency.
3. **Ease of use**. The library should be easily usable wherever possible, even at slight costs of efficiency.
4. **Reusability and Generosity**. The algorithms in SeqAn should be reusable and generic, even at small costs of efficiency.

Modern C++

C++ is sometimes described as a language that most people know only 20% of but everyone knows a different 20%. This section gives an overview over some C++ idioms we use. This might be no news if you are a seasoned C++ programmer who is apt at using the STL and Boost libraries. However, programmers coming from C and Java might find them interesting (We still encourage to read the C++ FAQ if you are new to C++).

References References are alternatives to pointers in C++ to construct value aliases. Also see Wikipedia on C++ references.

Templates C++ allows you to perform generic programming using templates. While similar to generics in Java (C++ templates are more than a decade older), C++ templates are designed to write zero-overhead abstractions that can be written to be as efficient as hand-written code while retaining a high level of abstraction. See cplusplus.com on C++ templates. Note that there is no way to restrict the type that can be used in templates, there is no mechanism such as Java's `? extends T` in C++. Using an incompatible type leads to compiler errors because some operator or function could not be found.

Memory Management / No Pointers Object oriented programming is another key programming paradigm made available with C++ (Compared to C). This means, that instead of using raw pointers to allocated chunks of memory, memory management should be done using containers. The STL provides containers such as std::vector and SeqAn offers String.

Memory Management in SeqAn

C++ allows to allocate complex objects on the stack (in contrast to Java where objects are always constructed on the heap). The objects are constructed when the code execution enters the scope/block they are defined in and freed when the block is left. Allocation of resources (e.g. memory) happens on construction and deallocation happens when the current block is left. This is best explained in an example.

```
#include <seqan/sequence.h>

int main(int argc, char const ** argv)
{
    seqan::String<char> programName = argv[0];
    if (argc > 1)
    {
        seqan::String<char> firstArg = argv[1];
        if (argc > 2)
            return 1;
    }
    return 0;
}
```

`seqan::String<char>` is a class (actually an instantiation of the class template String) that allows to store strings of `char` values, similar to `std::vector<char>` or `std::string`.

When the variable `programName` is allocated, the constructor of the `String<char>` class is called. It allocates sufficient memory to store the value of `argv[0]` and then copies over the values from this string. The variable exists until the current block is left. Since it is defined in the `main()` function, this can only happen in the last line of `main()` at the `return 0`. When the variable goes out of scope, its value is deconstructed and all allocated memory is freed.

If an argument was given to the program, the block in the `if` clause is entered. When this happens, the variable `firstArg` is constructed, memory is allocated and the value of `argv[1]` is copied into the buffer. When the block is left, the variable is deconstructed and all memory is deallocated.

Note that all memory is released when the `main()` function is left, regardless whether it is left in the `return 0` or the `return 1`. Corresponding code in C would be (arguably) more messy, either requiring `goto` or multiple `free()` calls, one before either `return`.

Motivation for Template Programming

In this section, we will give a short rationale why C++ with heavy use of template programming was used for SeqAn.

Any sequence analysis will have sequence data structures and algorithms on sequences at its heart. Even when only considering DNA and amino acid alphabets, there are various variants for alphabets that one has to consider. Otherwise, important applications in bioinformatics cannot be covered:

- 4-character DNA,
- 5-character DNA with `N`,
- 15-character IUPAC, and
- 27-character amino acids.

A simple implementation could simply store such strings as ASCII characters. However, there are some implementation tricks that can lead to great reduction of memory usage (e.g. encoding eight 4-character DNA characters in one byte) or running time (fast lookup tables for characters or q-grams) for small alphabets. Thus, simply using a `std::string` would come at high costs to efficiency.

Given that in the last 10-15 years, Java and C# have gained popularity, one could think about an object oriented solution: strings could simply be arrays of `Character` objects. Using polymorphism (e.g. overwriting of functions in subclasses), one could then write generic and reusable algorithms. For example, the Java 2 platform defines the sort function for all objects implementing a `Comparable` interface. Note that such an implementation would have to rely on virtual functions of some sort. However, as we will see in the section OOP vs. Generic Progamming, **this comes at a high performance cost, being in conflict with efficiency**. For a sequence library, we could implement functions that map values from an alphabet to an ordinal value between `0` and `S - 1` where `S` is the number of elements in the alphabet.

Generic programming offers one way out: C++ templates allow to define template classes, e.g. the STL's `std::vector<T>` or SeqAn's String. Here, instead of creating a string class around an array of `char` values (or objects), we can leave the type of the array's elements open. We can then introduce different types, e.g. `Dna` or `Dna5` for 4- and 5-character DNA alphabets.

Algorithms can be implemented using template functions and the template types are fixed at compile time. Thus, the compiler does not have to use virtual function tables and other "crutches", less indirection is involved, and more code can be inlined and aggressively optimized. When written appropriately, such algorithms can also work on different string implementations! Also, when defining our own alphabet types, we can directly influence how their abstractions (and APIs) work.

Thus, C++ allows us to implement (1) a generic and reusable library with (2) high level abstractions (and thus ease of use) that still allows the compiler to employ aggressive optimization and thus achieves (3) efficiency. With the words of the C++ inventor Bjarne Stroustrup:

> A high level of abstraction is good, not just in C++, but in general. We want to deal with problems at the level we are thinking about those problems. When we do that, we have no gap between the way we understand problems and the way we implement their solutions. We can understand the next guy's code. We don't have to be the compiler.

OOP vs. Generic Programming

In SeqAn, we use a technique called template subclassing which is based on generic programming. This technique provides polymorphism into C++ programs at **compile time** using templates. Such static polymorphism is different from **runtime polymorphism** which is supported in C++ using subclassing and virtual functions. It comes at the cost of some additional typing but has the advantage that the compiler can inline all function calls and thus achieve better performance. An example will be given in the section "From OOP to SeqAn" in the First Steps Tutorial.

Todo

We need a little code example here.

The important point is that in contrast to runtime polymorphism such static polymorphism allows the compiler to inline functions, which has huge effect on the overall performance of the program. Which as you recall correctly from above, is the main objective of the SeqAn library :)!

Global Function Interface

As we already stated, using template subclassing to achieve OOP like behavior in a more efficient way comes with a certain drawback. Subclassed objects are seen by the compiler as singular instances of a specific type. That means a subclassed object does not inherit the member or member functions of the alleged base class. In order to reduce the overhead of reimplementing the same member functions for every subclassed object, we use global interface functions.

You might already have get in touch with global function interfaces while working with the STL. With the new C++11 standard the STL now provides some global interface functions, e.g., the begin or end interface.

The rationale behind is the following observation. Global interface functions allow us to implement a general functionality that is used for all subclassed objects of this template class (assuming the accessed member variables exists in all subclassed objects as in the base template class, otherwise the compiler will complain). If the behavior for any subclassed object changes, the corresponding global function will be reimplemented for this special type covering the desired functionality. Due to template deduction the compiler already chooses the correct function and inlines the kernel if possible, which very likely improves the performance of the program. By this design, we can avoid code duplication, and by that increasing maintainability and reducing subtle errors due to less copy-and-paste code.

So, while most C++ developers, who are familiar with the STL and have a strong background in OO programming, are used to the typical dot notation, in SeqAn you have to get used to global function interfaces instead. But, cheer up! You will adapt to this very quickly. Promised!

Meta-Programming

Generic algorithms usually have to know certain types that correspond to their arguments. An algorithm on containers may need to know which type of values are stored in the string, or what kind of iterator we need to access it. The usual way in the STL is to define the value type of a class like `vector` as a *member typedef* of this class, so it can be retrieved by `vector::value_type`.

Unfortunately member typedef declarations have the same disadvantages as any members: Since they are specified by the class definition, they cannot be changed or added to the class without changing the code of the class, and it is not possible in C++ to define members for built-in types. What we need therefore is a mechanism that returns an output type (e.g. the value type) given an input type (e.g. the string) and doing so does not rely on members of the input type, but instead uses some kind of global interface.

Such task can be performed by **metafunctions**, also known as **type traits**. A metafunction is a construct to map some types or constants to other entities like types, constants, functions, or objects at compile time. The name metafunction comes from fact that they can be regarded as part of a meta-programming language that is evaluated during compilation.

In SeqAn we use class templates to implement metafunctions in C++. Generic algorithms usually have to know certain types that correspond to their arguments: An algorithm on strings may need to know which type of characters are stored in the string, or what kind of iterator can be used to browse it. SeqAn uses Metafunctions (also known as "traits") for that purpose.

Looking at an Example

Assuming that we define a string of amino acids:

```
    String<AminoAcid> amino_str = "ARN";
```

Now lets define a function that exchanges the first two values in a string:

```
void amino_exchangeFirstValues(String<AminoAcid> & str)
{
    if (length(str) < 2)
        return;
    AminoAcid temp = str[0];
    str[0] = str[1];
    str[1] = temp;
}
```

Since this function only works for instances of String<AminoAcid>, we could try to make it more general by making a template out of it.

```
template <typename T>
void general_exchangeFirstValues(T & str)
{
    if (length(str) < 2)
        return;
    AminoAcid temp = str[0];
    str[0] = str[1];
    str[1] = temp;
}
```

Now the function works for all sequence types `T` that store `AminoAcid` objects, but it will fail for other value types as soon as the variable temp cannot store `str[0]` anymore. To overcome this problem, we must redefine `temp` in a way that it can store a value of the correct type. The question is: "Given a arbitrary type `T`, what is the value type of `T`?"

The metafunction Value answers this question: "The value type of `T` is given by `Value<T>::Type`."

Hence, the final version of our function `exchangeFirstValues` reads as follows:

```
template <typename T>
void exchangeFirstValues(T & str)
{
    if (length(str) < 2)
        return;
    typename Value<T>::Type temp = str[0];
    str[0] = str[1];
    str[1] = temp;
}
```

We can view `Value` as a kind of "function" that takes `T` as an argument (in angle brackets) and returns the required value type of `T`. In fact, `Value` is not implemented as a C++ function, but as a class template. This class template is specialized for each sequence type `T` in a way that the `typedef Type` provides the value type of `T`. Unfortunately, the current C++ language standard does not allow to write simply "`Value<T> temp;`", so we must select the return value by appending "`::Type`". The leading "`typename`" becomes necessary since `Value<T>::Type` is a type that depends on a template parameter of the surrounding function template.

And now?

Wow, this was quite some information to digest, wasn't it? We suggest you take a break! Get some fresh air! Grab something to drink or to eat! Let the information settle down.

Do you think you've got everything? Well, if not don't worry! Follow the *First Steps* tutorial which will cover the topics discussed above. This gives you the chance to apply the recently discussed paradigms to an actual (uhm, simplistic) use case. But it will help you to better understand the way data structures and algorithms are implemented in SeqAn.

We recommend you to also read the *Argument Parser Tutorial*. This tutorial will teach you how to easily add command line arguments for your program and how to generate a help page for the options. Or you go back to the *main page* and stroll through the other tutorials. You are now ready to dive deeper into SeqAn. Enjoy!

ToC

Contents

A First Example

Learning Objective You will learn the most basic concepts of SeqAn. After this tutorial you will be ready to deal with the more specific tutorials, e.g. Sequences.

Difficulty Very basic

Duration 1.5h

Prerequisites Basic C or C++ knowledge

Welcome to the SeqAn "Hello World". This is the first practical tutorial you should look at when starting to use our software library.

We assume that you have some programming experience (preferably in C++ or C) and concentrate on SeqAn specific aspects. We will start out pretty slowly and hopefully the tutorial will make sense to you even if you are new to C++. However, to really leverage the power of SeqAn you will have to learn C++. There are many tutorials on C++, for example the tutorial at cplusplus.com.

This tutorial will walk you through a simple example program that highlights the things that are most prominently different from the libraries that many SeqAn newcomers are used to:

- extensive usage of C++ templates,
- generic programming using templates,
- using references instead of pointers in most places,
- and more.

Running Example

Let's start with a simple example programm. The program will do a pattern search of a short query sequence (pattern) in a long subject sequence (text). We define the score for each position of the database sequence as the sum of matching characters between the pattern and the text.

The following figure shows an expected result:

```
score:    101 ...        ... 801 ...
text:     This is an awesome tutorial to get to know SeqAn!
pattern:  tutorial           tutorial
           tutorial           tutorial
            ...                ...
```

The first position has a score of 1, because the `i` in the pattern matches the `i` in `is`. This is only a toy example for explanatory reasons and we ignore any more advanced implementations.

In SeqAn the program could look like this (we will explain every line of code shortly):

```
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

using namespace seqan;

int main()
{
    // Initialization
    String<char> text = "This is an awesome tutorial to get to know SeqAn!";
    String<char> pattern = "tutorial";

    String<int> score;
    resize(score, length(text) - length(pattern) + 1);

    // Computation of the similarities
    // Iteration over the text (outer loop)
    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
    {
        int localScore = 0;
        // Iteration over the pattern for character comparison
        for (unsigned j = 0; j < length(pattern); ++j)
        {
            if (text[i + j] == pattern[j])
                ++localScore;
        }
        score[i] = localScore;
    }

    // Printing the result
    for (unsigned i = 0; i < length(score); ++i)
        std::cout << score[i] << " ";
    std::cout << std::endl;

    return 0;
}
```

Whenever we use SeqAn classes or functions we have to explicitly write the namespace qualifier `seqan::` in front of the class name or function. This can be circumvented if we include the line `using namespace seqan;` at the top of the working example. However, during this tutorial we will not do this, such that SeqAn classes and functions can be recognized more easily.

Attention: Argument-Dependent Name Lookup (Koenig Lookup)

Using the namespace prefix `seqan::` is not really necessary in all places. In many cases, the Koenig lookup rule in C++ for functions makes this unnecessary. Consider the following, compiling, example.

```
seqan::String<char> s = "example";
unsigned i = length(s);
```

Here, the function `length` does not have a namespace prefix. The code compiles nevertheless. The compiler automatically looks for a function `length` in the namespace of its arguments.

Note that we follow the rules for variable, function, and class names as outlined in the *SeqAn style guide*. For example: 1. variables and functions use lower case, 2. struct, enum and classes use CamelCase, 3. metafunctions start with a capital letter, and 4. metafunction values are UPPERCASE.

Assignment 1

Type Review

Objective Create a demo program and replace its content with the code above.

Hint Depending on your operating system you have different alternatives to create a demo application. An in depth description can be found in GettingStarted.

Solution Click ''more...''

```
// Copy the code into a demo program and have a look at the result.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

using namespace seqan;

int main()
{
    // Initialization
    String<char> text = "This is an awesome tutorial to get to know SeqAn!";
    String<char> pattern = "tutorial";

    String<int> score;
    resize(score, length(text) - length(pattern) + 1);

    // Computation of the similarities
    // Iteration over the text (outer loop)
    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
    {
        int localScore = 0;
        // Iteration over the pattern for character comparison
        for (unsigned j = 0; j < length(pattern); ++j)
        {
            if (text[i + j] == pattern[j])
                ++localScore;
        }
        score[i] = localScore;
    }

    // Printing the result
    for (unsigned i = 0; i < length(score); ++i)
        std::cout << score[i] << " ";
    std::cout << std::endl;
```

```
    // > 1 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 8 0 1 0 0 0 0 2 0 1 0 0 1 0 3 0 1
↪0 1 0 0 0 0

    return 0;
}
```

SeqAn and Templates

Let us now have a detailed look at the program.

We first include the IOStreams library that we need to print to the screen and the SeqAn's `<seqan/file.h>` as well as `<seqan/sequence.h>` module from the SeqAn library that provides SeqAn String.

```
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

using namespace seqan;
```

The String class is one of the most fundamental classes in SeqAn, which comes as no surprise since SeqAn is used to analyse sequences (there is an extra tutorial for SeqAn *sequences* and *alphabets*).

In contrast to the popular string classes of Java or C++, SeqAn provides different string implementations and different alphabets for its strings. There is one string implementation that stores characters in memory, just like normal C++ strings. Another string implementation stores the characters on disk and only keeps a part of the sequence in memory. For alphabets, you can use strings of nucleotides, such as genomes, or you can use strings of amino acids, for example.

SeqAn uses **template functions** and **template classes** to implement the different types of strings using the **generic programming** paradigm. Template functions/classes are normal functions/classes with the additional feature that one passes the type of a variable as well as its value (see also: templates in cpp). This means that SeqAn algorithms and data structures are implemented in such a way that they work on all types implementing an informal interface (see information box below for more details). This is similar to the philosophy employed in the C++ STL (Standard Template Library).

The following two lines make use of template programming to define two strings of type char, a text and a pattern.

```
    // Initialization
    String<char> text = "This is an awesome tutorial to get to know SeqAn!";
    String<char> pattern = "tutorial";
```

In order to store the similarities between the pattern and different text positions we additionally create a string storing integer values.

```
    String<int> score;
```

Note that in contrast to the first two string definitions we do not know the values of the different positions in the string in advance. In order to dynamically adjust the length of the new string to the text we can use the function resize. The resize function is not a member function of the string class because SeqAn is not object oriented in the typical sence (we will see later how we adapt SeqAn to object oriented programming). Therefore, instead of writing `string.resize(newLength)` we use `resize(string, newLength)`.

```
    resize(score, length(text) - length(pattern) + 1);
```

Note: Global function interfaces.

SeqAn uses **global interfaces** for its data types/classes. Generally, you have to use `function(variable)` instead of `variable.function()`.

This has the advantage that we can extend the interface of a type outside of its definition. For example, we can provide a `length()` function for STL containers `std::string<T>` and `std::vector<T>` outside their class files. We can use such global functions to make one data type have the same interface as a second. This is called **adaption**.

Additionally, we can use one function definition for several data types. For example, the alignment algorithms in SeqAn are written such that we can compute alignments using any String with any alphabet: There are more than 5 String variants in SeqAn and more than 8 built-in alphabets. Thus, one implementation can be used for more than 40 different data types!

After the string initializations it is now time for the similarity computation. In this toy example we simply take the pattern and shift it over the text from left to right. After each step, we check how many characters are equal between the corresponding substring of the text and the pattern. We implement this using two loops; the outer one iterates over the given text and the inner loop over the given pattern:

```
    // Computation of the similarities
    // Iteration over the text (outer loop)
    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
    {
        int localScore = 0;
        // Iteration over the pattern for character comparison
        for (unsigned j = 0; j < length(pattern); ++j)
        {
            if (text[i + j] == pattern[j])
                ++localScore;
        }
        score[i] = localScore;
    }
```

There are two things worth mentioning here: (1) SeqAn containers or strings start at position 0 and (2) you will notice that we use `++variable` instead of `variable++` wherever possible. The reason is that `++variable` is slightly faster than its alternative, since the alternative needs to make a copy of itself before returning the result.

In the last step we simply print the result that we stored in the variable ```` ```score ```` on screen. This gives the similarity of the pattern to the string at each position.

```
    // Printing the result
    for (unsigned i = 0; i < length(score); ++i)
        std::cout << score[i] << " ";
    std::cout << std::endl;
```

Refactoring

At this point, we have already created a working solution! However, in order to make it easier to maintain and reuse parts of the code we need to export them into functions. In this example the interesting piece of code is the similarity computation, which consists of an outer and inner loop. We encapsulate the outer loop in function `computeScore` and the inner loop in function `computeLocalScore` as can be seen in the following code.

```
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

using namespace seqan;
```

```
int computeLocalScore(String<char> subText, String<char> pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;

    return localScore;
}

String<int> computeScore(String<char> text, String<char> pattern)
{
    String<int> score;
    resize(score, length(text) - length(pattern) + 1, 0);

    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + length(pattern)), pattern);

    return score;
}

int main()
{
    String<char> text = "This is an awesome tutorial to get to know SeqAn!";
    String<char> pattern = "tutorial";
    String<int> score = computeScore(text, pattern);

    for (unsigned i = 0; i < length(score); ++i)
        std::cout << score[i] << " ";
    std::cout << std::endl;

    return 0;
}
```

The function computeScore() now contains the fundamental part of the code and can be reused by other functions. The input arguments are two strings. One is the pattern itself and one is a substring of the text. In order to obtain the substring we can use the function infix implemented in SeqAn. The function call `infix(text, i, j)` generates a substring equal to `text[i ... j - 1]`, e.g. `infix(text, 1, 5)` equals "ello", where `text` is "Hello World". To be more precise, infix() generates a Infix which can be used as a string, but is implemented using pointers such that no copying is necessary and running time and memory is saved.

Assignment 2

Type Review

Objective Replace the code in your current file by the code above and encapsulate the print instructions.

Hint The function head should look like this:

```
void print(String<int> text)
```

Solution

```
// Copy the code into your current file and encapsulate the print instructions.

#include <iostream>
#include <seqan/file.h>
```

```
#include <seqan/sequence.h>

using namespace seqan;

int computeLocalScore(String<char> subText, String<char> pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;

    return localScore;
}

String<int> computeScore(String<char> text, String<char> pattern)
{
    String<int> score;
    resize(score, length(text) - length(pattern) + 1, 0);

    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + length(pattern)),␣
↪pattern);

    return score;
}

//![head]
void print(String<int> text)
//![head]
{
    for (unsigned i = 0; i < length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

int main()
{
    String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    String<char> pattern = "tutorial";
    String<int> score = computeScore(text, pattern);

    print(score);

    return 0;
}
```

The Role of References in SeqAn

Let us now have a closer look at the signature of `computeScore()`.

Both the text and the pattern are passed *by value*. This means that both the text and the pattern are copied when the function is called, which consumes twice the memory. This can become a real bottleneck since copying longer sequences is very memory and time consuming, think of the human genome, for example.

Instead of copying we could use **references**. A reference in C++ is created using an ampersand sign (`&`) and creates an alias to the referenced value. Basically, a reference is a pointer to an object which can be used just like the referenced object itself. This means that when you change something in the reference you also change the original object it came

from. But there is a solution to circumvent this modification problem as well, namely the word **const**. A `const` object cannot be modified.

Important: If an object does not need to be modified make it an nonmodifiably object using the keyword `const`. This makes it impossible to *unwillingly* change objects, which can be really hard to debug. Therefore it is recommended to use it as often as possible.

Therefore we change the signature of computeScore to:

```
String<int> computeScore(String<char> const & text, String<char> const & pattern)
```

Reading from right to left the function expects two `references` to `const objects` of type `String` of `char`.

Assignment 3

Type Review

Objective Adjust your current code to be more memory and time efficient by using references in the function header.

Hint The function head for `computeLocalScore` should look like this:

```
int computeLocalScore(String<char> const & subText, String<char> const & pattern)
```

Solution

```
// Adjust your current code to be more memory and time efficient by using␣
↪references in the function header.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>

using namespace seqan;

//![head_local]
int computeLocalScore(String<char> const & subText, String<char> const & pattern)
//![head_local]
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;

    return localScore;
}

//![head]
String<int> computeScore(String<char> const & text, String<char> const & pattern)
//![head]
{
    String<int> score;
    resize(score, length(text) - length(pattern) + 1, 0);

    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + length(pattern)),␣
↪pattern);
```

```
    return score;
}

void print(String<int> const & text)
{
    for (unsigned i = 0; i < length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

int main()
{
    String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    String<char> pattern = "tutorial";
    String<int> score = computeScore(text, pattern);

    print(score);

    return 0;
}
```

Generic and Reusable Code

As mentioned earlier, there is another issue: the function computeScore only works for Strings having the alphabet `char`. If we wanted to use it for `Dna` or `AminoAcid` strings then we would have to reimplement it even though the only difference is the signature of the function. All used functions inside `computeScore` can already handle the other datatypes.

The more appropriate solution is a generic design using templates, as often used in the SeqAn library. Instead of specifying the input arguments to be references of strings of `char` s we could use references of template arguments as shown in the following lines:

```
template <typename TText, typename TPattern>
String<int> computeScore(TText const & text, TPattern const & pattern)
```

The first line above specifies that we create a template function with two template arguments `TText` and `TPattern`. At compile time the template arguments are then replace with the correct types. If this line was missing the compiler would expect that there are types `TText` and `TPattern` with definitions.

Now the function signature is better in terms of memory consumption, time efficiency, and generality.

Important: The SeqAn Style Guide

The *SeqAn style guide* gives rules for formatting and structuring C++ code as well as naming conventions. Such rules make the code more consistent, easier to read, and also easier to use.

1. **Naming Scheme**. Variable and function names are written in `lowerCamelCase`, type names are written in `UpperCamelCase`. Constants and enum values are written in `UPPER_CASE`. Template variable names always start with 'T'.
2. **Function Parameter Order**. The order is (1) output, (2) non-const input (e.g. file handles), (3) input, (4) tags. Output and non-const input can be modified, the rest is left untouched and either passed by copy or by const-reference (`const &`).
3. **Global Functions**. With the exception of constructors and a few operators that have to be defined in-class, the interfaces in SeqAn use global functions.

4. **No Exceptions**. The SeqAn interfaces do not throw any exceptions.

While we are trying to make the interfaces consistent with our style guide, some functions have incorrect parameter order. This will change in the near future to be more in line with the style guide.

Assignment 4

Type Review

Objective Generalize the `computeLocalScore` function in your file.

Solution

```
// Generalize the computeLocalScore function in you file.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>
#include <seqan/score.h>

using namespace seqan;

template <typename TText, typename TPattern>
int computeLocalScore(TText const & subText, TPattern const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;

    return localScore;
}

template <typename TText, typename TPattern>
String<int> computeScore(TText const & text, TPattern const & pattern)
{
    String<int> score;
    resize(score, length(text) - length(pattern) + 1, 0);

    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + length(pattern)),␣
↪pattern);

    return score;
}

void print(String<int> const & text)
{
    for (unsigned i = 0; i < length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

int main()
{
    String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    String<char> pattern = "tutorial";
```

```
    String<int> score = computeScore(text, pattern);
    print(score);
    return 0;
}
```

From Object-Oriented Programming to SeqAn

There is another huge advantage of using templates: we can specialize a function without touching the existing function. In our working example it might be more appropriate to treat `AminoAcid` sequences differently. As you probably know, there is a similarity relation on amino acids: Certain amino acids are more similar to each other, than others. Therefore we want to score different kinds of mismatches differently. In order to take this into consideration we simple write a `computeLocalScore()` function for `AminoAcid` strings. In the future whenever 'computerScore' is called always the version above is used unless the second argument is of type String-AminoAcid. Note that the second template argument was removed since we are using the specific type String-AminoAcid.

```
template <typename TText>
int computeLocalScore(TText const & subText, seqan::String<seqan::AminoAcid> const &␣
↪pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < seqan::length(pattern); ++i)
        localScore += seqan::score(seqan::Blosum62(), subText[i], pattern[i]);

    return localScore;
}
```

In order to score a mismatch we use the function `score()` from the SeqAn library. Note that we use the Blosum62 matrix as a similarity measure. When looking into the documentation of score you will notice that the score function requires a argument of type Score. This object tells the function how to compare two letters and there are several types of scoring schemes available in SeqAn (of course, you can extend this with your own). In addition, because they are so frequently used there are shortcuts as well. For example Blosum62 is really a **shortcut** for `Score<int, ScoreMatrix<AminoAcid, Blosum62_> >`, which is obviously very helpful. Other shortcuts are `DnaString` for `String<Dna>` (*sequence tutorial*), `CharString` for `String<char>`, ...

Tip: Template Subclassing

The main idea of template subclassing is to exploit the C++ template matching mechanism. For example, in the following code, the function calls (1) and (3) will call the function `myFunction()` in variant (A) while the function call (2) will call variant (B).

```
struct SpecA;
struct SpecB;
struct SpecC;

template <typename TAlphabet, typename TSpec>
class String{};

template <typename TAlphabet, typename TSpec>
void myFunction(String<TAlphabet, TSpec> const &){}  // Variant (A)

template <typename TAlphabet>
void myFunction(String<TAlphabet, SpecB> const &){}  // Variant (B)

// ...
```

```
int main()
{
    String<char, SpecA> a;
    String<char, SpecB> b;
    String<char, SpecC> c;

    myFunction(a);            // calls (A)
    myFunction(b);            // calls (B)
    myFunction(c);            // calls (A)
}
```

Assignment 5

Type Application

Objective Provide a generic print function which is used when the input type is not `String<int>`.

Hint Keep your current implementation and add a second function. Don't forget to make both template functions. Include `<seqan/score.h>` as well.

Solution

```
// Provide a generic print function which is used when the input type is not␣
↪String<int>.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>
#include <seqan/score.h>

using namespace seqan;

template <typename TText>
int computeLocalScore(TText const & subText, String<AminoAcid> const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        localScore += score(Blosum62(), subText[i], pattern[i]);

    return localScore;
}

template <typename TText, typename TPattern>
int computeLocalScore(TText const & subText, TPattern const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;

    return localScore;
}

template <typename TText, typename TPattern>
String<int> computeScore(TText const & text, TPattern const & pattern)
{
```

```
    String<int> score;
    resize(score, length(text) - length(pattern) + 1, 0);

    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + length(pattern)),␣
↪pattern);

    return score;
}

template <typename TText>
void print(TText const & text)
{
    std::cout << text << std::endl;
}

void print(String<int> const & text)
{
    for (unsigned i = 0; i < length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

int main()
{
    String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    String<char> pattern = "tutorial";
    String<int> score = computeScore(text, pattern);

    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(pattern);
    // > tutorial
    print(score);
    // > 1 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 8 0 1 0 0 0 0 2 0 1 0 0 1 0 3 0 1␣
↪1 0 0 0 0
    return 0;
}
```

Tags in SeqAn

Sometimes you will see something like this:

```
    globalAlignment(align, seqan::MyersHirschberg());
```

Having a closer look you will notice that there is a default constructor call (`MyersHirschberg()`) within a function call. Using this mechanism one can specify which function to call at compile time. The `MyersHirschberg()` '' is only a tag to determine which specialisation of the `globalAligment` function to call.

If you want more information on tags then read on otherwise you are now ready to explore SeqAn in more detail and continue with one of the other tutorials.

There is another use case of templates and function specialization.

This might be useful in a `print()` function, for example. In some scenarios, we only want to print the position where the maximal similarity between pattern and text is found. In other cases, we might want to print the similarities of all

positions. In SeqAn, we use **tag-based dispatching** to realize this. Here, the type of the **tag** holds the specialization information.

Tip: Tag-Based Dispatching

You will often see **tags** in SeqAn code, e.g. `Standard()`. These are parameters to functions that are passed as const-references. They are not passed for their values but for their type only. This way, we can select different specializations at **compile time** in a way that plays nicely together with metafunctions, template specializations, and an advanced technique called [[Tutorial/BasicTechniques| metaprogramming]].

Consider the following example:

```
template <typename T>
struct Tag{};

struct TagA_;
typedef Tag<TagA_> TagA;

struct TagB_;
typedef Tag<TagB_> TagB;

void myFunction(TagA const &){}  // (1)
void myFunction(TagB const &){}  // (2)

int main()
{
    myFunction(TagA());  // (3)
    myFunction(TagB());  // (4)
    return 0;
}
```

The function call in line (3) will call `myFunction()` in the variant in line (1). The function call in line (4) will call `myFunction()` in the variant in line (2).

The code for the two different `print()` functions mentioned above could look like this:

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/score.h>

template <typename TText, typename TSpec>
void print(TText const & text, TSpec const & /*tag*/)
{
    for (unsigned i = 0; i < seqan::length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

struct MaxOnly {};

template <typename TText>
void print(TText const & score, MaxOnly const & /*tag*/)
{
    int maxScore = score[0];
    seqan::String<int> output;
    appendValue(output, 0);
    for (unsigned i = 1; i < seqan::length(score); ++i)
```

```
    {
        if (score[i] > maxScore)
        {
            maxScore = score[i];
            clear(output);
            resize(output, 1, i);
        }
        else if (score[i] == maxScore)
        {
            appendValue(output, i);
        }
    }

    for (unsigned i = 0; i < seqan::length(output); ++i)
        std::cout << output[i] << " ";
    std::cout << std::endl;
}

int main()
{
    return 0;
}
```

If we call `print()` with something different than `MaxOnly` then we print all the positions with their similarity, because the generic template function accepts anything as the template argument. On the other hand, if we call print with `MaxOnly` only the positions with the maximum similarity as well as the maximal similarity will be shown.

Assignment 6

Type Review

Objective Provide a print function that prints pairs of positions and their score if the score is greater than 0.

Hints SeqAn provides a data type Pair.

Solution

```
// Provide a print function that prints pairs of positions and their score if the␣
↪score is greater than 0.

#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>
#include <seqan/score.h>

using namespace seqan;

template <typename TText>
int computeLocalScore(TText const & subText, String<AminoAcid> const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        localScore += score(Blosum62(), subText[i], pattern[i]);

    return localScore;
}
```

```
template <typename TText, typename TPattern>
int computeLocalScore(TText const & subText, TPattern const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;

    return localScore;
}

template <typename TText, typename TPattern>
String<int> computeScore(TText const & text, TPattern const & pattern)
{
    String<int> score;
    resize(score, length(text) - length(pattern) + 1, 0);

    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + length(pattern)),␣
↪pattern);

    return score;
}

template <typename TText>
void print(TText const & text)
{
    std::cout << text << std::endl;
}

void print(String<int> const & text)
{
    for (unsigned i = 0; i < length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

template <typename TText, typename TSpec>
void print(TText const & text, TSpec const & /*tag*/)
{
    print(text);
}

struct MaxOnly {};

template <typename TText>
void print(TText const & score, MaxOnly const & /*tag*/)
{
    int maxScore = score[0];
    String<int> output;
    appendValue(output, 0);
    for (unsigned i = 1; i < length(score); ++i)
    {
        if (score[i] > maxScore)
        {
            maxScore = score[i];
            clear(output);
            resize(output, 1, i);
```

```
        }
        else if (score[i] == maxScore)
            appendValue(output, i);
    }

    print(output);
}

struct GreaterZero {};

template <typename TText>
void print(TText const & score, GreaterZero const & /*tag*/)
{
    String<Pair<int> > output;
    for (unsigned i = 1; i < length(score); ++i)
        if (score[i] > 0)
            appendValue(output, Pair<int>(i, score[i]));

    for (unsigned i = 0; i < length(output); ++i)
        std::cout << "(" << output[i].i1 << "; " << output[i].i2 << ") ";
    std::cout << std::endl;
}

int main()
{
    String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    String<char> pattern = "tutorial";
    String<int> score = computeScore(text, pattern);

    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(pattern);
    // > tutorial
    print(score);
    // > 1 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 8 0 1 0 0 0 0 2 0 1 0 0 1 0 3 0 1␣
↪1 0 0 0 0
    print(score, MaxOnly());
    // > 19
    print(score, GreaterZero());
    // > (2; 1) (5; 1) (12; 1) (17; 1) (19; 8) (21; 1) (26; 2) (28; 1) (31; 1)␣
↪(33; 3) (35; 1) (36; 1)

    // And now for a protein pattern
    String<AminoAcid> protein = "tutorial";
    String<int> proteinScore = computeScore(text, protein);

    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(protein);
    // > TXTXRIAL
    print(proteinScore);
    // > 6 -9 -3 -6 -6 0 -9 -8 -7 -3 -9 -5 -8 -4 -5 -6 -6 1 -6 25 -7 2 -6 -6 -9 -
↪6 -5 -7 1 -7 -5 -4 -6 2 -6 -3 -8 -9 -10 -4 -6 0 0 0 0 0 0 0
    print(proteinScore, MaxOnly());
    // > 19
    print(proteinScore, GreaterZero());
    // > (17; 1) (19; 25) (21; 2) (28; 1) (33; 2)
```

```
    return 0;
}
```

Obviously this is only a toy example in which we could have named the two `print()` functions differently. However, often this is not the case when the programs become more complex. Because SeqAn is very generic we do not know the datatypes of template functions in advance. This would pose a problem because the function call of function `b()` in function `a()` may depend on the data types of the template arguments of function `a()`.

The Final Result

Don't worry if you have not fully understood the last section. If you have – perfect. In any case the take home message is that you use data types for class specializations and if you see a line of code in which the default constructor is written in a function call this typical means that the data type is important to distinct between different function implementations.

Now you are ready to explore more of the SeqAn library. There are several tutorials which will teach you how to use the different SeqAn data structures and algorithms. Below you find the complete code for our example with the corresponding output.

```
#include <iostream>
#include <seqan/file.h>
#include <seqan/sequence.h>
#include <seqan/score.h>

using namespace seqan;

template <typename TText>
int computeLocalScore(TText const & subText, String<AminoAcid> const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        localScore += score(Blosum62(), subText[i], pattern[i]);

    return localScore;
}

template <typename TText, typename TPattern>
int computeLocalScore(TText const & subText, TPattern const & pattern)
{
    int localScore = 0;
    for (unsigned i = 0; i < length(pattern); ++i)
        if (subText[i] == pattern[i])
            ++localScore;

    return localScore;
}

template <typename TText, typename TPattern>
String<int> computeScore(TText const & text, TPattern const & pattern)
{
    String<int> score;
    resize(score, length(text) - length(pattern) + 1, 0);

    for (unsigned i = 0; i < length(text) - length(pattern) + 1; ++i)
        score[i] = computeLocalScore(infix(text, i, i + length(pattern)), pattern);
```

```
    return score;
}

template <typename TText>
void print(TText const & text)
{
    std::cout << text << std::endl;
}

void print(String<int> const & text)
{
    for (unsigned i = 0; i < length(text); ++i)
        std::cout << text[i] << " ";
    std::cout << std::endl;
}

template <typename TText, typename TSpec>
void print(TText const & text, TSpec const & /*tag*/)
{
    print(text);
}

struct MaxOnly {};

template <typename TText>
void print(TText const & score, MaxOnly const & /*tag*/)
{
    int maxScore = score[0];
    String<int> output;
    appendValue(output, 0);
    for (unsigned i = 1; i < length(score); ++i)
    {
        if (score[i] > maxScore)
        {
            maxScore = score[i];
            clear(output);
            resize(output, 1, i);
        }
        else if (score[i] == maxScore)
            appendValue(output, i);
    }

    print(output);
}

struct GreaterZero {};

template <typename TText>
void print(TText const & score, GreaterZero const & /*tag*/)
{
    String<Pair<int> > output;
    for (unsigned i = 1; i < length(score); ++i)
        if (score[i] > 0)
            appendValue(output, Pair<int>(i, score[i]));

    for (unsigned i = 0; i < length(output); ++i)
        std::cout << "(" << output[i].i1 << "; " << output[i].i2 << ") ";
    std::cout << std::endl;
```

```
}

int main()
{
    String<char> text = "This is an awesome tutorial to get to now SeqAn!";
    String<char> pattern = "tutorial";
    String<int> score = computeScore(text, pattern);

    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(pattern);
    // > tutorial
    print(score);
    // > 1 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 8 0 1 0 0 0 0 2 0 1 0 0 1 0 3 0 1 1 0␣
↪0 0 0
    print(score, MaxOnly());
    // > 19
    print(score, GreaterZero());
    // > (2; 1) (5; 1) (12; 1) (17; 1) (19; 8) (21; 1) (26; 2) (28; 1) (31; 1) (33;␣
↪3) (35; 1) (36; 1)

    // And now for a protein pattern
    String<AminoAcid> protein = "tutorial";
    String<int> proteinScore = computeScore(text, protein);

    print(text);
    // > This is an awesome tutorial to get to now SeqAn!
    print(protein);
    // > TXTXRIAL
    print(proteinScore);
    // > 6 -9 -3 -6 -6 0 -9 -8 -7 -3 -9 -5 -8 -4 -5 -6 -6 1 -6 25 -7 2 -6 -6 -9 -6 -5␣
↪-7 1 -7 -5 -4 -6 2 -6 -3 -8 -9 -10 -4 -6 0 0 0 0 0 0 0
    print(proteinScore, MaxOnly());
    // > 19
    print(proteinScore, GreaterZero());
    // > (17; 1) (19; 25) (21; 2) (28; 1) (33; 2)

    return 0;
}
```

ToC

Contents

Parsing Command Line Arguments

Learning Objective You will learn how to use the ArgumentParser class to parse command line arguments. This tutorial is a walk-through with links into the API documentation and also meant as a source for copy-and-paste code.

Difficulty Easy

Duration 30-60 min

Prerequisites *A First Example*, *Sequences*, familiarity with building SeqAn apps

The simplest possible and also most flexible interface to a program is through the command line. This tutorial explains how to parse the command line using the SeqAn library's ArgumentParser class.

Using this class will give you the following functionality:

- Robust parsing of command line arguments.
- Simple verification of arguments (e.g. within a range, one of a list of allowed values).
- Automatically generated and nicely formatted help screens when called with `--help`. You can also export this help to HTML and man pages.
- You are able to automatically generate nodes for workflow engines such as KNIME or Galaxy.

As a continuous example, we will write a little program that is given strings on the command line and applies an operation to every i-th character:

```
# modify_string --uppercase -i 2 "This is some text!"
ThIs iS SoMe TeXt!
# modify_string "This is some text!" --lowercase -i 1
this is some text!
```

The program has three types of command line options/arguments:

- Two **flag options** `--uppercase` and `--lowercase` that select the operation.
- One **(value) option** `-i` that selects the period of the characters that the operation is to be applied to and is given a value (`2` in the first call above, `1` in the second).
- One **(positional) argument** with the text to modify (`"This is some text!"` in both calls above. In contrast to options, arguments are not identified by their names but by their position.

Command line options can have a **long name** (e.g. `--lowercase`) and/or a **short name** (e.g. `-i`).

A First Working Example

The following small program will (1) setup a ArgumentParser object named `parser`, (2) parse the command line, (3) exit the program if there were errors or the user requested a functionality that is already built into the command line parser, and (4) printing the settings given from the command line. Such functionality is printing the help, for example.

```
#include <iostream>

#include <seqan/arg_parse.h>

int main(int argc, char const ** argv)
{
```

```
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    // Extract option values and print them.
    unsigned period = 0;
    getOptionValue(period, parser, "period");
    bool toUppercase = isSet(parser, "uppercase");
    seqan::CharString text;
    getArgumentValue(text, parser, 0);

    std::cout << "period   \t" << period << '\n'
              << "uppercase\t" << toUppercase << '\n'
              << "text     \t" << text << '\n';

    return 0;
}
```

Let us first play a bit around with the program before looking at it in detail.

For example, we can already let the program generate an online help:

```
# modify_string -h
modify_string
=============

SYNOPSIS

DESCRIPTION
    -h, --help
          Displays this help message.
    -i, --period INT
          Period to use for the index.
    -U, --uppercase
          Select to-uppercase as operation.

VERSION
    modify_string version:
    Last update
```

While already informative, the help screen looks like there is something missing. For example, there is no synopsis, no version and no date of the last update given. We will fill this in later.

When we pass some parameters, the settings are printed:

```
# modify_string "This is a test." -i 1 -U
period     1
uppercase  1
text       This is a test.
```

When we try to use the `--lowercase`/`-L` option, we get an error. This is not surprising since we did not tell the argument parser about this option yet.

```
# modify_string "This is a test." -i 1 -L
modify_string: illegal option -- L
```

A Detailed Look

Let us look at this program in detail now. The required SeqAn module is `seqan/arg_parse.h`. After inclusion, we can create an ArgumentParser object:

```
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");
```

Then, we define a positional argument using the function addArgument. The function accepts the parser and an ArgParseArgument object. We call the ArgParseArgument constructor with two parameters: the type of the argument (a string), and a label for the documentation.

```
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));
```

Then, we add options to the parser using addOption. We pass the parser and an ArgParseOption object.

```
    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
```

The ArgParseOption constructor is called in two different variants. Within the first addOption call, we construct an integer option with a short and long name, a documentation string, and give it the label "INT". The second option is a flag (indicated by not giving a type) with a short and a long name and a description.

Next, we parse the command line using parse.

```
    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);
```

We then check the result of the parsing operation. The result is `seqan::ArgumentParser::PARSE_ERROR` if there was a problem with the parsing. Otherwise, it is `seqan::ArgumentParser::PARSE_OK` if there was no problem and no special functionality of the argument parser was triggered. The command line parser automatically adds some arguments, such as `--help`. If such built-in functionality is triggered, it will return a value that is neither `PARSE_ERROR` nor `PARSE_OK`.

The following two lines have the following behaviour. If the parsing went through and no special functionality was triggered then the branch is not taken. Otherwise, the method `main()` is left with `1` in case of errors and with `0` in case special behaviour was triggered (e.g. the help was printed).

```
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;
```

Finally, we access the values from the command line using the ArgumentParser. The function getOptionValue allows us to access the values from the command line after casting into C++ types. The function isSet allows us to query whether a given argument was set on the command line.

```
    // Extract option values and print them.
    unsigned period = 0;
    getOptionValue(period, parser, "period");
    bool toUppercase = isSet(parser, "uppercase");
    seqan::CharString text;
    getArgumentValue(text, parser, 0);

    std::cout << "period   \t" << period << '\n'
              << "uppercase\t" << toUppercase << '\n'
              << "text     \t" << text << '\n';
```

Tip: List Arguments and Options.

You have to mark an option to be a list if you want to be able to collect multiple values for it from the command line. Consider the following program call:

```
# program -a 1 -a 2 -a 3
```

If the option `a` is not a list then the occurence `-a 3` overwrites all previous settings.

However, if `a` is marked to be a list, then all values (`1`, `2`, and `3`) are stored as its values. We can get the number of elements using the function getOptionValueCount and then access the individual arguments using the function getOptionValue. You can mark an option and arguments to be lists by using the `isList` parameter to the ArgParseArgument and ArgParseOption constructors.

For arguments, only the first or the last argument or none can be a list but not both. Consider this program call:

```
# program arg0 arg1 arg2 arg3
```

For example, if the program has three arguments and the first one is a list then `arg0` and `arg1` would be the content of the first argument. If it has two arguments and the last one is a list then `arg1`, `arg2`, and `arg3` would be the content of the last argument.

Assignment 1

Type Reproduction

Objective Copy the source code of the full First Working Example above into a demo. Compile it and test printing the help screen and calling it with the two command lines above.

Solution You can do it!

Assignment 2

Type Reproduction

Objective Adjust the program from above to also accept an option to convert characters to lower case, just as it accepts options to convert characters to upper case. The long name should be `--lowercase`, the short name should be `-L`. As for the `--uppercase` option, the program should print whether the flag was set or not.

Hint Copy the two lines for defining the `--uppercase` option and replace the strings appropriately.

Solution

```
#include <iostream>

#include <seqan/arg_parse.h>

int main(int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    addArgument(parser, seqan::ArgParseArgument(
    seqan::ArgParseArgument::STRING, "TEXT"));

    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    // Extract option values and print them.
    unsigned period = 0;
    getOptionValue(period, parser, "period");
    bool toUppercase = isSet(parser, "uppercase");
    bool toLowercase = isSet(parser, "lowercase");
    seqan::CharString text;
    getArgumentValue(text, parser, 0);

    std::cout << "period   \t" << period << '\n'
              << "uppercase\t" << toUppercase << '\n'
              << "lowercase\t" << toLowercase << '\n'
              << "text     \t" << text << '\n';

    return 0;
}
```

Using Default Values

Would it not be nice if we could specify a default value for `--period`, so it is `1` if not specified and simply each character is modified? We can do this by using the function setDefaultValue:

```
setDefaultValue(parser, "period", "1");
```

Note that we are giving the default value as a string. The ArgumentParser object will simply interpret it as if it was given on the command line. There, of course, each argument is a string.

Assignment 3

Setting a default value

Type Reproduction

Objective Adjust the previous program to accept default values by adding the `setDefaultValue()` line from above into your program.

Solution

```
#include <iostream>

#include <seqan/arg_parse.h>

int main(int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    // Extract option values and print them.
    unsigned period = 0;
    getOptionValue(period, parser, "period");
    bool toUppercase = isSet(parser, "uppercase");
    bool toLowercase = isSet(parser, "lowercase");
    seqan::CharString text;
    getArgumentValue(text, parser, 0);

    std::cout << "period   \t" << period << '\n'
              << "uppercase\t" << toUppercase << '\n'
              << "lowercase\t" << toLowercase << '\n'
              << "text     \t" << text << '\n';
```

```
    return 0;
}
```

Best Practice: Using Option Structs

Instead of just printing the options back to the user, we should actually store them. To follow best practice, we should not use global variables for this but instead pass them as parameters.

We will thus create a `ModifyStringOptions` struct that encapsulates the settings the user can give to the `modify_string` program. Note that we initialize the variables of the struct with initializer lists, as it is best practice in modern C++.

The `ModifyStringOptions` struct's looks as follows:

```
struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
    period(1), toUppercase(false), toLowercase(false)
    {}
};
```

Click **more...** to see the whole updated program.

```
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
    period(1), toUppercase(false), toLowercase(false)
    {}
};

int main(int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setDefaultValue(parser, "period", "1");
```

```
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    // Extract option values and print them.
    ModifyStringOptions options;
    getOptionValue(options.period, parser, "period");
    options.toUppercase = isSet(parser, "uppercase");
    options.toLowercase = isSet(parser, "lowercase");
    getArgumentValue(options.text, parser, 0);

    std::cout << "period   \t" << options.period << '\n'
              << "uppercase\t" << options.toUppercase << '\n'
              << "lowercase\t" << options.toLowercase << '\n'
              << "text     \t" << options.text << '\n';

    return 0;
}
```

Best Practice: Wrapping Parsing In Its Own Function

As a next step towards a cleaner program, we should extract the argument parsing into its own function, e.g. call it `parseCommandLine()`. Following the style guide (*C++ Code Style*), we first pass the output parameter, then the input parameters. The return value of our function is a `seqan::ArgumentParser::ParseResult` such that we can differentiate whether the program can go on, the help was printed and the program is to exit with success, or there was a problem with the passed argument and the program is to exit with an error code.

Also, note that we should check that the user cannot specify both to-lowercase and to-uppercase. This check cannot be performed by the ArgumentParser by itself but we can easily add this check. We add this functionality to the `parseCommandLine()` function.

Click **more...** to see the updated program.

```
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
    period(1), toUppercase(false), toLowercase(false)
    {}
};
```

```
seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    // Define Options
    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // Only extract  options if the program will continue after parseCommandLine()
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res;

    // Extract option values.
    getOptionValue(options.period, parser, "period");
    options.toUppercase = isSet(parser, "uppercase");
    options.toLowercase = isSet(parser, "lowercase");
    getArgumentValue(options.text, parser, 0);

    // If both to-uppercase and to-lowercase were selected then this is an error.
    if (options.toUppercase && options.toLowercase)
    {
        std::cerr << "ERROR: You cannot specify both to-uppercase and to-lowercase!\n
↪";
        return seqan::ArgumentParser::PARSE_ERROR;
    }

    return seqan::ArgumentParser::PARSE_OK;
}


int main(int argc, char const ** argv)
{
    // Parse the command line.
    ModifyStringOptions options;
    seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    std::cout << "period   \t" << options.period << '\n'
```

```
              << "uppercase\t" << options.toUppercase << '\n'
              << "lowercase\t" << options.toLowercase << '\n'
              << "text     \t" << options.text << '\n';

    return 0;
}
```

Feature-Complete Example Program

The command line parsing part of our program is done now. Let us now add a function `modifyText()` that is given a `ModifyStringOptions` object and text and modifies the text. We simply use the C standard library functios `toupper()` and `tolower()` from the header `<cctype>` for converting to upper and lower case.

```
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
    period(1), toUppercase(false), toLowercase(false)
    {}
};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    // Define Options
    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // Only extract  options if the program will continue after parseCommandLine()
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res;
```

```
    // Extract option values.
    getOptionValue(options.period, parser, "period");
    options.toUppercase = isSet(parser, "uppercase");
    options.toLowercase = isSet(parser, "lowercase");
    getArgumentValue(options.text, parser, 0);

    // If both to-uppercase and to-lowercase were selected then this is an error.
    if (options.toUppercase && options.toLowercase)
    {
        std::cerr << "ERROR: You cannot specify both to-uppercase and to-lowercase!\n
↪";
        return seqan::ArgumentParser::PARSE_ERROR;
    }

    return seqan::ArgumentParser::PARSE_OK;
}

seqan::CharString modifyString(seqan::CharString const & text,
                               ModifyStringOptions const & options)
{
    seqan::CharString result;

    if (options.toLowercase)
    {
        for (unsigned i = 0; i < length(text); ++i)
        {
            if (i % options.period == 0u)
                appendValue(result, tolower(text[i]));
            else
                appendValue(result, text[i]);
        }
    }
    else
    {
        for (unsigned i = 0; i < length(text); ++i)
        {
            if (i % options.period == 0u)
                appendValue(result, toupper(text[i]));
            else
                appendValue(result, text[i]);
        }
    }

    return result;
}

int main(int argc, char const ** argv)
{
    // Parse the command line.
    ModifyStringOptions options;
    seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc, argv);

    // If parsing was not successful then exit with code 1 if there were errors.
    // Otherwise, exit with code 0 (e.g. help was printed).
    if (res != seqan::ArgumentParser::PARSE_OK)
        return res == seqan::ArgumentParser::PARSE_ERROR;

    std::cout << modifyString(options.text, options) << '\n';
```

```
    return 0;
}
```

Setting Restrictions

One nice feature of the ArgumentParser is that it is able to perform some simple checks on the parameters. We can:

- check numbers for whether they are greater/smaller than some limits,
- mark options as being required, and
- setting lists of valid values for each option.

In this section, we will give some examples.

Setting Minimum and Maximum Values

The functions setMinValue and setMaxValue allow to give a smallest and/or largest value for a given option. Of course, this only works with integer- and double-typed command line options.

We can pass both the short and the long option name to these functions. The value is given as a string and parsed the same as parameters on the command line.

```
seqan::ArgumentParser parser("modify_string");
addOption(parser, seqan::ArgParseOption(
    "i", "integer-value", "An integer option",
    seqan::ArgParseArgument::INTEGER, "INT"));

setMinValue(parser, "i", "10");
setMaxValue(parser, "integer-value", "20");
```

Assignment 4

Setting min-value on `--period`

Type Reproduction

Objective Use the function setMinValue to set a minimal value of `1` for the parameter `--period`.

Solution

```
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    unsigned rangeBegin, rangeEnd;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
        period(1), rangeBegin(0), rangeEnd(0),toUppercase(false),
```

```
        toLowercase(false)
    {}
};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
        // Setup ArgumentParser.
        seqan::ArgumentParser parser("modify_string");

        // We require one argument.
        addArgument(parser, seqan::ArgParseArgument(
            seqan::ArgParseArgument::STRING, "TEXT"));

        // Define Options
        addOption(parser, seqan::ArgParseOption(
            "i", "period", "Period to use for the index.",
            seqan::ArgParseArgument::INTEGER, "INT"));
        setMinValue(parser, "period", "1");
        setDefaultValue(parser, "period", "1");
        addOption(parser, seqan::ArgParseOption(
            "r", "range", "Range of the text to modify.",
            seqan::ArgParseArgument::INTEGER, "INT", false, 2));
        addOption(parser, seqan::ArgParseOption(
            "U", "uppercase", "Select to-uppercase as operation."));
        addOption(parser, seqan::ArgParseOption(
            "L", "lowercase", "Select to-lowercase as operation."));

        // Parse command line.
        seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

        // Only extract  options if the program will continue after␣
↪parseCommandLine()
        if (res != seqan::ArgumentParser::PARSE_OK)
            return res;

        // Extract option values.
        getOptionValue(options.period, parser, "period");
        getOptionValue(options.rangeBegin, parser, "range", 0);
        getOptionValue(options.rangeEnd, parser, "range", 1);
        options.toUppercase = isSet(parser, "uppercase");
        options.toLowercase = isSet(parser, "lowercase");
        seqan::getArgumentValue(options.text, parser, 0);

        // If both to-uppercase and to-lowercase were selected then this is an␣
↪error.
        if (options.toUppercase && options.toLowercase)
        {
            std::cerr << "ERROR: You cannot specify both to-uppercase and to-
↪lowercase!\n";
            return seqan::ArgumentParser::PARSE_ERROR;
        }

        return seqan::ArgumentParser::PARSE_OK;
}

seqan::CharString modifyString(seqan::CharString const & text,
                               ModifyStringOptions const & options)
```

```
{
        seqan::CharString result;

        if (options.toLowercase)
        {
            for (unsigned i = 0; i < length(text); ++i)
            {
                if (i >= options.rangeBegin && i < options.rangeEnd &&
                    (i % options.period == 0u))
                    appendValue(result, tolower(text[i]));
                else
                    appendValue(result, text[i]);
            }
        }
        else
        {
            for (unsigned i = 0; i < length(text); ++i)
            {
                if (i >= options.rangeBegin && i < options.rangeEnd &&
                    (i % options.period == 0u))
                    appendValue(result, toupper(text[i]));
                else
                    appendValue(result, text[i]);
            }
        }

        return result;
}

int main(int argc, char const ** argv)
{
        // Parse the command line.
        ModifyStringOptions options;
        seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc,␣
↪argv);

        // If parsing was not successful then exit with code 1 if there were␣
↪errors.
        // Otherwise, exit with code 0 (e.g. help was printed).
        if (res != seqan::ArgumentParser::PARSE_OK)
            return res == seqan::ArgumentParser::PARSE_ERROR;

        std::cout << modifyString(options.text, options) << '\n';

        return 0;
}
```

Marking Options as Required

We can mark options as being required using the function setRequired:

```
seqan::ArgumentParser parser("modify_string");
addOption(parser, seqan::ArgParseOption(
    "i", "integer-value", "An integer option",
    seqan::ArgParseArgument::INTEGER, "INT"));
```

```
setRequired(parser, "i");
```

Setting List of Valid Values

Sometimes, it is useful to give a list of valid values for a command line option. You can give it as a space-separated list in a string to setValidValues. The check whether the value from the command line is valid is case sensitive.

```
seqan::ArgumentParser parser("modify_string");
addOption(parser, seqan::ArgParseOption(
    "", "distance-model", "Distance model, either HAMMING or EDIT.",
    seqan::ArgParseArgument::STRING, "STR"));

setValidValues(parser, "distance-model", "HAMMING EDIT");
```

More Option and Argument Types

There are two slightly more special option and argument types: paths to input/output files and tuple values.

Input/Output File Names

We could use `ArgParseArgument::STRING` to specify input and output files. However, there are two special argument/option types `ArgParseArgument::INPUT_FILE` and `ArgParseArgument::OUTPUT_FILE` that are more suitable:

1. In the near future, we plan to add basic checks for whether input files exist and are readable by the user. You will still have to check whether opening was successful when actually doing this but the program will fail earlier if the source file or target location are not accessible. The user will not have to wait for the program to run through to see that he mistyped the output directory name, for example, and you do not have to write this check.
2. For workflow engine integration, the input and output file options and arguments will be converted into appropriate input and output ports of the nodes.
3. You can use the previously introduced restrictions to specify what kind of files you expect and the ArgumentParser will check while parsing if the correct file type was provided.

Here is an example for defining input and output file arguments:

```
addOption(parser, seqan::ArgParseOption(
    "I", "input-file", "Path to the input file",
    seqan::ArgParseArgument::INPUT_FILE, "IN"));
addOption(parser, seqan::ArgParseOption(
    "O", "output-file", "Path to the output file",
    seqan::ArgParseArgument::OUTPUT_FILE, "OUT"));
```

The restrictions are added by defining the expected file extension.

```
setValidValues(parser, "input-file", "txt");
setValidValues(parser, "output-file", "txt");
```

Again multiple values are provided as space-separated list. Note that the file ending check is case insensitive, so you do not need to provide `txt` and `TXT`.

You can simply read the values of these options as you would read string options:

```
seqan::CharString inputFileName, outputFileName;
seqan::getOptionValue(inputFileName, parser, "input-file");
seqan::getOptionValue(outputFileName, parser, "output-file");
```

Assignment 5

Using File Command Line Options

Type Reproduction

Objective Replace the argument `TEXT` by a command line option `-I/--input-file` in the program above. The program should then read in the text instead of using the command line argument.

Hint We will also replace the `text` member of `ModifyStringOptions`, you might wish to do the same.

Solution

```
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    unsigned rangeBegin, rangeEnd;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString inputFileName;

    ModifyStringOptions() :
        period(1), rangeBegin(0), rangeEnd(0),toUppercase(false),
        toLowercase(false)
    {}
};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // Define Options
    addOption(parser, seqan::ArgParseOption(
        "I", "input-file",
        "A text file that will printed with the modifications applied.",
        seqan::ArgParseArgument::INPUT_FILE));
    setValidValues(parser, "input-file", "txt");
    setRequired(parser, "input-file");

    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setMinValue(parser, "period", "1");
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));
```

```
        // Parse command line.
        seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

        // Only extract  options if the program will continue after␣
↪parseCommandLine()
        if (res != seqan::ArgumentParser::PARSE_OK)
            return res;

        // Extract option values.
        getOptionValue(options.period, parser, "period");
        options.toUppercase = isSet(parser, "uppercase");
        options.toLowercase = isSet(parser, "lowercase");
        getOptionValue(options.inputFileName, parser, "input-file");

        // If both to-uppercase and to-lowercase were selected then this is an␣
↪error.
        if (options.toUppercase && options.toLowercase)
        {
            std::cerr << "ERROR: You cannot specify both to-uppercase and to-
↪lowercase!\n";
            return seqan::ArgumentParser::PARSE_ERROR;
        }

        return seqan::ArgumentParser::PARSE_OK;
    }

    seqan::CharString modifyString(seqan::CharString const & text,
                                   ModifyStringOptions const & options)
    {
        seqan::CharString result;

        if (options.toLowercase)
        {
            for (unsigned i = 0; i < length(text); ++i)
            {
                if (i % options.period == 0u)
                    appendValue(result, tolower(text[i]));
                else
                    appendValue(result, text[i]);
            }
        }
        else
        {
            for (unsigned i = 0; i < length(text); ++i)
            {
                if (i % options.period == 0u)
                    appendValue(result, toupper(text[i]));
                else
                    appendValue(result, text[i]);
            }
        }

        return result;
    }

    int main(int argc, char const ** argv)
    {
```

```
        // Parse the command line.
        ModifyStringOptions options;
        seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc,␣
↪argv);

        // If parsing was not successful then exit with code 1 if there were␣
↪errors.
        // Otherwise, exit with code 0 (e.g. help was printed).
        if (res != seqan::ArgumentParser::PARSE_OK)
            return res == seqan::ArgumentParser::PARSE_ERROR;

        std::fstream inFile(toCString(options.inputFileName), std::ios::binary |␣
↪std::ios::in);
        if (inFile.good())
        {
            std::cerr << "ERROR: Could not open input file " << options.
↪inputFileName << '\n';
            return 1;
        }
        seqan::CharString text;
        while (inFile.good())
        {
            char c = inFile.get();
            if (inFile.good())
                appendValue(text, c);
        }
        std::cout << modifyString(text, options);

        return 0;
    }
```

Tuples

We can define an ArgParseArgument and ArgParseOption to be a tuple with a fixed number of arguments. For example, an integer pair (tuple with two entries) could describe a range:

```
addOption(parser, seqan::ArgParseOption(
    "r", "range", "The range to modify.",
    seqan::ArgParseArgument::INTEGER, "BEGIN END",
    false, 2));
```

We add two parameters after the label `"BEGIN END"` for the documentation. First, we specify that the option is not a list option (`false`) and second, that we need exactly two numbers for it.

The user can now use the parameter as follows:

```
# modify_string -r 5 10 ...
```

We use the four-parameter variant with an integer index of getOptionValue to access the entries in the tuple given on the command line.

```
unsigned rangeBegin = 0, rangeEnd = 0;
getOptionValue(rangeBegin, parser, "range", 0);
getOptionValue(rangeEnd, parser, "range", 1);
```

Assignment 6

Using Tuple Command Line Options

Type Reproduction

Objective Add a command line option `--range` to the ArgumentParser in the program above. Modify the function `modifyString()` such that only parameters in the given range are changed.

Hint We will add two `unsigned` members `rangeBegin` and `rangeEnd` to the `ModifyStringOptions` struct, you might wish to do the same.

Solution

```
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    unsigned rangeBegin, rangeEnd;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
        period(1), rangeBegin(0), rangeEnd(0),toUppercase(false),
        toLowercase(false)
    {}
};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    // Define Options
    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setMinValue(parser, "period", "1");
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));

    // Parse command line.
    seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

    // Only extract  options if the program will continue after␣
↪parseCommandLine()
    if (res != seqan::ArgumentParser::PARSE_OK)
```

```
            return res;

        // Extract option values.
        getOptionValue(options.period, parser, "period");
        options.toUppercase = isSet(parser, "uppercase");
        options.toLowercase = isSet(parser, "lowercase");
        seqan::getArgumentValue(options.text, parser, 0);

        // If both to-uppercase and to-lowercase were selected then this is an␣
↪error.
        if (options.toUppercase && options.toLowercase)
        {
            std::cerr << "ERROR: You cannot specify both to-uppercase and to-
↪lowercase!\n";
            return seqan::ArgumentParser::PARSE_ERROR;
        }

        return seqan::ArgumentParser::PARSE_OK;
    }

    seqan::CharString modifyString(seqan::CharString const & text,
                                   ModifyStringOptions const & options)
    {
        seqan::CharString result;

        if (options.toLowercase)
        {
            for (unsigned i = 0; i < length(text); ++i)
            {
                if (i % options.period == 0u)
                    appendValue(result, tolower(text[i]));
                else
                    appendValue(result, text[i]);
            }
        }
        else
        {
            for (unsigned i = 0; i < length(text); ++i)
            {
                if (i % options.period == 0u)
                    appendValue(result, toupper(text[i]));
                else
                    appendValue(result, text[i]);
            }
        }

        return result;
    }

    int main(int argc, char const ** argv)
    {
        // Parse the command line.
        ModifyStringOptions options;
        seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc,␣
↪argv);

        // If parsing was not successful then exit with code 1 if there were␣
↪errors.
```

```
        // Otherwise, exit with code 0 (e.g. help was printed).
        if (res != seqan::ArgumentParser::PARSE_OK)
            return res == seqan::ArgumentParser::PARSE_ERROR;

        std::cout << modifyString(options.text, options) << '\n';

        return 0;
    }
```

Embedding Rich Documentation

Another very useful feature of ArgumentParser is that you can embed rich documentation into your programs. You can set the short description, the version string, date, synopsis and add text documentation settings.

Let us first set the **short description**, **version string**, and **date** in our program from above. We insert the following lines just after the declaration of the variable `parser`.

```
setShortDescription(parser, "String Modifier");
setVersion(parser, "1.0");
setDate(parser, "July 2012");
```

After the line with `setDate()`, we give a usage line and add to the description. This information will go to the Synopsis section of the program help.

```
addUsageLine(parser,
             "[\\fIOPTIONS\\fP] \"\\fITEXT\\fP\"");
addDescription(parser,
               "This program allows simple character modifications to "
               "each i-th character.");
```

Tip: Formatting Command Line Documentation

The formatting of command line parameters might seem strange, at first: **Font operators** start with `\f` (which means that they start with `"\\f"` in in C++ string literals). The `\\f` is followed by the **format specifier**. The format specifier can be one of `I`, `B`, and `P`. `I` selects italic text (underlined on the shell), `B` selects bold and `P` resets the formatting to normal text. These font operators are legacies of man pages from Unix and offered a simple-to-implement solution to text formatting.

For example, `"Words \\fBwere\\fP made for \\fIbeing\\fP written!"` would result in the formatted string "Words **were** made for *being* written!".

Note that formatting the command line relies on ANSI escape codes which is not supported by modern Windows versions. If you are using Windows, you will not see bold or underlined text.

The argument parser will add some options of its own, for example for printing the help and displaying version information. To separate our arguments from the autogenerated ones, we add the following line. This line will introduce the section "Modification Options" in the Description section of the output.

```
addSection(parser, "Modification Options");
```

Finally, we will add a section with examples. Add the following lines just before the line with the `parse()` function call.

```
addTextSection(parser, "Examples");

```

```
addListItem(parser,
            "\\fBmodify_string\\fP \\fB-U\\fP \\fIveryverylongword\\fP",
            "Print upper case version of \"veryverylongword\"");
addListItem(parser,
            "\\fBmodify_string\\fP \\fB-L\\fP \\fB-i\\fP \\fI3\\fP␣
↪\\fIveryverylongword\\fP",
            "Print \"veryverylongword\" with every third character "
            "converted to upper case.");
```

That were a lot of changes! Click **more...** to see the complete program.

```
#include <iostream>

#include <seqan/arg_parse.h>

struct ModifyStringOptions
{
    unsigned period;
    bool toUppercase;
    bool toLowercase;
    seqan::CharString text;

    ModifyStringOptions() :
        period(1), toUppercase(false), toLowercase(false)
    {}
};

seqan::ArgumentParser::ParseResult
parseCommandLine(ModifyStringOptions & options, int argc, char const ** argv)
{
    // Setup ArgumentParser.
    seqan::ArgumentParser parser("modify_string");
    // Set short description, version, and date.
    setShortDescription(parser, "String Modifier");
    setVersion(parser, "1.0");
    setDate(parser, "July 2012");

    // Define usage line and long description.
    addUsageLine(parser,
                 "[\\fIOPTIONS\\fP] \"\\fITEXT\\fP\"");
    addDescription(parser,
                   "This program allows simple character modifications to "
                   "each i-th character.");

    // We require one argument.
    addArgument(parser, seqan::ArgParseArgument(
        seqan::ArgParseArgument::STRING, "TEXT"));

    // Define Options -- Section Modification Options
    addSection(parser, "Modification Options");
    addOption(parser, seqan::ArgParseOption(
        "i", "period", "Period to use for the index.",
        seqan::ArgParseArgument::INTEGER, "INT"));
    setDefaultValue(parser, "period", "1");
    addOption(parser, seqan::ArgParseOption(
        "U", "uppercase", "Select to-uppercase as operation."));
    addOption(parser, seqan::ArgParseOption(
        "L", "lowercase", "Select to-lowercase as operation."));
```

```
        // Add Examples Section.
        addTextSection(parser, "Examples");
        addListItem(parser,
                    "\\fBmodify_string\\fP \\fB-U\\fP \\fIveryverylongword\\fP",
                    "Print upper case version of \"veryverylongword\"");
        addListItem(parser,
                    "\\fBmodify_string\\fP \\fB-L\\fP \\fB-i\\fP \\fI3\\fP "
                    "\\fIveryverylongword\\fP",
                    "Print \"veryverylongword\" with every third character "
                    "converted to upper case.");

        // Parse command line.
        seqan::ArgumentParser::ParseResult res = seqan::parse(parser, argc, argv);

        // Only extract  options if the program will continue after␣
↪parseCommandLine()
        if (res != seqan::ArgumentParser::PARSE_OK)
            return res;

        // Extract option values.
        getOptionValue(options.period, parser, "period");
        options.toUppercase = isSet(parser, "uppercase");
        options.toLowercase = isSet(parser, "lowercase");
        seqan::getArgumentValue(options.text, parser, 0);

        // If both to-uppercase and to-lowercase were selected then this is an␣
↪error.
        if (options.toUppercase && options.toLowercase)
        {
            std::cerr << "ERROR: You cannot specify both to-uppercase and to-
↪lowercase!\n";
            return seqan::ArgumentParser::PARSE_ERROR;
        }

        return seqan::ArgumentParser::PARSE_OK;
    }

    seqan::CharString modifyString(seqan::CharString const & text,
                                   ModifyStringOptions const & options)
    {
        seqan::CharString result;

        if (options.toLowercase)
        {
            for (unsigned i = 0; i < length(text); ++i)
                appendValue(result, tolower(text[i]));
        }
        else
        {
            for (unsigned i = 0; i < length(text); ++i)
                appendValue(result, toupper(text[i]));
        }

        return result;
    }

    int main(int argc, char const ** argv)
```

```
    {
        // Parse the command line.
        ModifyStringOptions options;
        seqan::ArgumentParser::ParseResult res = parseCommandLine(options, argc,␣
↪argv);

        // If parsing was not successful then exit with code 1 if there were errors.
        // Otherwise, exit with code 0 (e.g. help was printed).
        if (res != seqan::ArgumentParser::PARSE_OK)
            return res == seqan::ArgumentParser::PARSE_ERROR;

        std::cout << modifyString(options.text, options) << '\n';

        return 0;
    }
```

Let us look at the resulting documentation. Simply call the new program with the `--help` option.

```
# modify_string --help
modify_string - String Modifier
===============================

SYNOPSIS
    modify_string [OPTIONS] "TEXT"

DESCRIPTION
    This program allows simple character modifications to each
    i-th character.

    -h, --help
          Displays this help message.
    --version
          Display version information

  Modification Options:
    -i, --period INT
          Period to use for the index.
    -U, --uppercase
          Select to-uppercase as operation.
    -L, --lowercase
          Select to-lowercase as operation.

EXAMPLES
    modify_string -U veryverylongword
          Print upper case version of "veryverylongword"
    modify_string -L -i 3 veryverylongword
          Print "veryverylongword" with every third character
          converted to upper case.

VERSION
    modify_string version: 1.0
    Last update July 2012
```

Also, there is an undocumented option called `--export-help` that is automatically added by ArgumentParser. You can call it with the values `html` and `man`. If the option is set then the argument parser will print the documentation as HTML or man format (man pages are a widely used format for Unix documentation).

You can pipe the output to a file:

```
# modify_string --export-help html > modify_string.html
# modify_string --export-help man > modify_string.man
```

HTML can be displayed by any web browser, man pages can be displayed using the program `man`. Note that when opening a file using `man`, you have to give the file name either as an absolute or a relative path. Otherwise, it would try to look up the topic `modify_string.man`. To view the generated man page use:

```
# man ./modify_string.man
```

Below, you can see a part of the rendered HTML and man pages generated by the commands above.

For further deading, have a look at the ArgumentParser class.

Data Structures

Sequences

ToC

Contents

Strings and Segments

Learning Objective You will learn about the SeqAn sequence concept and its main class String as well as the class Segment. After completing this tutorial, you will be able to use important functionalities of sequences in SeqAn and you will be ready to continue with the more specific tutorials, e.g. *Alignment*, or *Pairwise Sequence Alignment*.

Difficulty Very basic

Duration 45 min

Prerequisites Basic C or C++ knowledge, the *A First Example* tutorial helps.

Sequences are the core concept of SeqAn. A sequence is a container that stores an ordered list of values. In SeqAn, there are three kinds of sequences: Strings, Sequence Adaptions and Segments.

The String class is one of the most fundamental classes in SeqAn. It is designed as a generic data structure that can be instantiated for all kinds of values, both simple (e.g. `char`, Dna, AminoAcid) and non-simple value types (e.g. Tuple, String). With sequence adaptions, SeqAn offers an interface for accessing data types that are not part of SeqAn, namely standard library strings and c-style char arrays. Thus those built-in types can be handled in a similar way as SeqAn strings, for example with the length function. Segments are contiguous subsequences that represent parts of other sequences.

This tutorial will deal with the SeqAn sequence classes String and Segment.

Strings

In this section, we will have a detailed look at the SeqAn class String. You will learn how to build and expand strings as well as how to compare and convert them.

Defining Strings

Let's first have a look at a simple example on how to define a String. The type of the contained value is specified by the first template argument, e.g. `char` or `int`.

```
String<char>  myText;     // A string of characters.
String<int>   myNumbers;  // A string of integers.
```

To fill the string with contents, we can simply assign a string literal to the created variable:

```
myText = "Hello SeqAn!";
```

Any type that provides a default constructor, a copy constructor and an assignment operator can be used as the alphabet / contained type of a String. This includes the C++ POD types, e.g. `char`, `int`, `double` etc., or even more complex types complex types, such as Strings.

```
String<String<char> >   myStringList;   // A string of character strings.
```

Hint: Nested Sequences (aka "Strings of Strings")

A collection of sequences can either be stored in a sequence of sequences, for example in a `String< String<char> >`, or in a StringSet. The latter one allows for more auxiliary functionalities to improve the efficiency of working with large sequence collections. You can learn more about it in the tutorial *String Sets*.

SeqAn also provides the following types that are useful in bioinformatics: AminoAcid, Dna, Dna5, DnaQ, Dna5Q, Finite, Iupac, Rna, Rna5. You can find detailed information in the tutorial *Alphabets*.

```
String<Dna>         myGenome;   // A string of nucleotides.
String<AminoAcid>   myProtein;  // A string of amino acids.
```

For commonly used string parameterizations, SeqAn has a range of shortcuts implemented, e.g. DnaString, RnaString and Peptide.

```
// Instead of String<Dna> dnaSeq we can also write:
DnaString dnaSeq = "TATA";
```

Working with Strings

The SeqAn String implementation provides the common C++ operators that you know already from the vector class of the STL. For example:

```
String<Dna> dnaSeq = "TATA";
dnaSeq += "CGCG";
std::cout << dnaSeq << std::endl;
```

```
TATACGCG
```

Each sequence object has a capacity, i.e. the maximum length of a sequence that can be stored in this object. While some sequence types have a fixed capacity, the capacity of other sequence classes like Alloc String or `std::basic_string` can be changed at runtime. The capacity can be set explicitly by functions such as reserve or resize. It can also be set implicitly by functions like append or replace, if the operation's result exceeds the length of the target string.

In the following example we create a String of Dna5String. We first set the new length of the container with resize to two elements. After assigning two elements we append one more element with appendValue. In the last step the capacity is implicitly changed.

```
    String<Dna5String> readList;
    resize(readList, 2);
    readList[0] = "GGTTTCGACG";
    readList[1] = "AAGATGTCGC";
    appendValue(readList, "TATGCATGAT");
```

Using the function length, we can now get the length of our strings, e.g.:

```
    std::cout << length(readList) << std::endl;
    std::cout << length(readList[0]) << std::endl;
```

```
3
10
```

To empty a String, the function clear resets the object.

```
    clear(readList);
```

SeqAn offers a range of other functions for the work with the String class, e.g. assign, assignValue, value, getValue, empty, etc. The full list of functions you can find in the documentation String.

Assignment 1

Type Review

Objective In the following assignment, you will write a small function that builds the reverse complement of a given string. Copy the code below and add the following functionalities:

1. Use the `resize` function to `resize` the `revComplGenome` variable.
2. Using the `getRevCompl` function, get the reverse complement for every nucleotide `genome` and store it in reverse order `revComplGenome`.
3. Print out the original genome and the reverse complement.

```
#include <seqan/sequence.h>
#include <seqan/basic.h>
#include <seqan/stream.h>
#include <seqan/file.h>
#include <seqan/modifier.h>

using namespace seqan;

Dna getRevCompl(Dna const & nucleotide)
{
    if (nucleotide == 'A')
        return 'T';
    if (nucleotide == 'T')
        return 'A';
    if (nucleotide == 'C')
        return 'G';
    return 'C';
}
```

```
int main()
{
    DnaString genome = "TATATACGCGCGAGTCGT";
    DnaString revComplGenome;
```

```
// Your code snippet here
```

```
    // And to check if your output is correct,
    // use the given SeqAn function reverseComplement(),
    // which modifies the sequence in-place:
    reverseComplement(genome);
    std::cout << genome << std::endl;
    return 0;
}
```

Hints Remember that the last element in `genome` is stored at position `length(genome) - 1`.

Solution Click *more...* to see the solution.

```
#include <seqan/sequence.h>
#include <seqan/basic.h>
#include <seqan/stream.h>
#include <seqan/file.h>
#include <seqan/modifier.h>

using namespace seqan;

Dna getRevCompl(Dna const & nucleotide)
{
    if (nucleotide == 'A')
        return 'T';
    if (nucleotide == 'T')
        return 'A';
    if (nucleotide == 'C')
        return 'G';
    return 'C';
}

int main()
{
    DnaString genome = "TATATACGCGCGAGTCGT";
    DnaString revComplGenome;
    //1.
    resize(revComplGenome, length(genome));
    //2.
    for (unsigned i = 0; i < length(genome); ++i)
        revComplGenome[length(genome) - 1 - i] = getRevCompl(genome[i]);
    //3.
    std::cout << genome << std::endl;
    std::cout << revComplGenome << std::endl;

    // And to check if your output is correct,
    // use the given SeqAn function reverseComplement(),
    // which modifies the sequence in-place:
    reverseComplement(genome);
    std::cout << genome << std::endl;
    return 0;
}
```

Your output should look like this:

```
TATATACGCGCGAGTCGT
ACGACTCGCGCGTATATA
ACGACTCGCGCGTATATA
```

Assignment 2

Type Review

Objective In this assignment, you will do some simple string building tasks, and write a simple alignment of the given reads and chromosomes. Use the given code template to solve these subtasks:

1. Assume we have mapped the reads to the positions 7, 100, 172, and 272 in 'chr1'. Store these positions in another string 'alignPosList'.
2. Build another String bsChr1 as a copy of chr1, and exchange every 'C' with a 'T', as in a bisulfite treated genome.
3. Print alignments of the reads and chr1 (or bschr1) using the function `printAlign` and the string `alignPosList`.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;
// Function to print simple alignment between two sequences with the same␣
↪length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout <<  "Alignment " << std::endl;
    std::cout << "  genome : " << genomeFragment << std::endl;
    std::cout << "  read   : " << read << std::endl;
}

int main()
{
    // Build reads and genomes
    DnaString chr1 =
↪"TATAATATTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTG
↪";

    // Build List containing all reads
    typedef String<DnaString> TDnaList;
    TDnaList readList;
    resize(readList, 4);
    readList[0] = "TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT
↪";
    readList[1] =
↪"TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT";
    readList[2] =
↪"AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACA
↪";
    readList[3] = "CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA
↪";
```

```
    // Append a second chromosome sequence fragment to chr1
    DnaString chr2 =
↪"AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACACGTCTCTGTGTTCCGACG
↪";
    append(chr1, chr2);

    // Print readlist
    std::cout << " \n Read list: " << std::endl;
    for (unsigned i = 0; i < length(readList); ++i)
        std::cout << readList[i] << std::endl;

    // 1. Assume we have mapped the 4 reads to chr1 (and chr2) and now have␣
↪the mapping start positions (no gaps).
    // Store the start position in a String alignPosList: 7, 100, 172, 272
```

```
// Your code snippet here for 1.+2.
```

```
    // 3. Print alignments of the reads with chr1 (or bsChr1) sequence using␣
↪the function printAlign
    // and the positions in alignPosList.
    // To do that, you have to create a copy of the fragment in chr1␣
↪(bsChr1) that is aligned to the read.
    std::cout << " \n Print alignment: " << std::endl;
    for (unsigned i = 0; i < length(readList); ++i)
    {
        // Begin position beginPosition of a given alignment between the␣
↪read and the genome
```

```
// Your code snippet here for 3.
```

```
        // Genome fragment
        DnaString genomeFragment;
```

```
// Your code snippet here for 3.
```

```
        // Call of our function to print the simple alignment
        printAlign(genomeFragment, readList[i]);
    }
    return 0;
}
```

Hints You have to create a copy of the fragment in chr1 (bsChr1) that is aligned to the read.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;
// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout <<  "Alignment " << std::endl;
```

```
    std::cout << "  genome : " << genomeFragment << std::endl;
    std::cout << "  read   : " << read << std::endl;
}

int main()
{
    // Build reads and genomes
    DnaString chr1 =
↪"TATAATATTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTG
↪";

    // Build List containing all reads
    typedef String<DnaString> TDnaList;
    TDnaList readList;
    resize(readList, 4);
    readList[0] = "TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT";
    readList[1] =
↪"TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT";
    readList[2] =
↪"AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGGTTCATGTGCGCTGAAGCACACATGCACA";
    readList[3] = "CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA";

    // Append a second chromosome sequence fragment to chr1
    DnaString chr2 =
↪"AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGGTTCATGTGCGCTGAAGCACACATGCACACGTCTCTGTGTTCCGACG
↪";
    append(chr1, chr2);

    // Print readlist
    std::cout << " \n Read list: " << std::endl;
    for (unsigned i = 0; i < length(readList); ++i)
        std::cout << readList[i] << std::endl;

    // 1. Assume we have mapped the 4 reads to chr1 (and chr2) and now have the␣
↪mapping start positions (no gaps).
    // Store the start position in a String alignPosList: 7, 100, 172, 272
    String<unsigned> alignPosList;
    resize(alignPosList, 4);
    alignPosList[0] = 7;
    alignPosList[1] = 100;
    alignPosList[2] = 172;
    alignPosList[3] = 272;

    // 2. Bisulfite conversion
    // Assume chr1 is beeing bisulfate treated: Copy chr1 to a new genome bsChr1␣
↪and exchange every 'C' with a 'T'
    DnaString bsChr1;
    assign(bsChr1, chr1);
    for (unsigned i = 0; i < length(bsChr1); ++i)
        if (bsChr1[i] == 'C')
            bsChr1[i] = 'T';
    // 3. Print alignments of the reads with chr1 (or bsChr1) sequence using the␣
↪function printAlign
    // and the positions in alignPosList.
    // To do that, you have to create a copy of the fragment in chr1 (bsChr1)␣
↪that is aligned to the read.
    std::cout << " \n Print alignment: " << std::endl;
    for (unsigned i = 0; i < length(readList); ++i)
```

```
    {
        // Begin position beginPosition of a given alignment between the read and␣
↪the genome
        unsigned beginPosition = alignPosList[i];

        // Genome fragment
        DnaString genomeFragment;

        // We have to create a copy of the corresponding fragment of the genome,␣
↪where the read aligns to
        for (unsigned j = 0; j < length(readList[i]); ++j)
            appendValue(genomeFragment, chr1[beginPosition + j]);

        // Call of our function to print the simple alignment
        printAlign(genomeFragment, readList[i]);
    }
    return 0;
}
```

```
 Read list:
TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT
TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT
AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGGTTCATGTGCGCTGAAGCACACATGCACA
CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA

 Print alignment:
Alignment
  genome : TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT
  read   : TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT
Alignment
  genome : TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT
  read   : TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT
Alignment
  genome :␣
↪AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGGTTCATGTGCGCTGAAGCACACATGCACA
  read   :␣
↪AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGGTTCATGTGCGCTGAAGCACACATGCACA
Alignment
  genome : CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA
  read   : CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA
```

Comparisons

Two sequences can be lexicographically **compared** using standard operators such as < or >=.

```
    String<char> a = "beta";
    String<char> b = "alpha";

    std::cout << (a != b) << std::endl;
    std::cout << (a < b) << std::endl;
    std::cout << (a > b) << std::endl;
```

```
1
0
```

```
1
```

Each comparison involves a scan of the two sequences for searching the first mismatch between the strings. This could be costly if the two sequences share a long common prefix. Suppose we want to branch in a program depending on whether `a < b`, `a == b`, or `a > b`.

```
if (a < b)       { /* code for case "a < b"  */ }
else if (a > b)  { /* code for case "a > b"  */ }
else             { /* code for case "a == b" */ }
```

In this case, although only one scan would be enough to decide what case is to be applied, each operator > and < performs a new comparison. SeqAn offers the class Lexical to avoid unnecessary sequence scans. Lexicals can store the result of a comparison, for example:

```
// Compare a and b and store the result in comp
Lexical<> comp(a, b);

if (isLess(comp))          { /* code for case "a < b"  */ }
else if (isGreater(comp)) { /* code for case "a > b"  */ }
else                       { /* code for case "a == b" */ }
```

Conversions

A sequence of type A values can be converted into a sequence of type B values, if A can be converted into B. SeqAn offers different conversion alternatives.

Copy conversion. The source sequence is copied into the target sequence. This can be done by assignment (`operator=`) or using the function assign.

```
String<Dna> dna_source = "acgtgcat";
String<char> char_target;
assign(char_target, dna_source);
std::cout << char_target << std::endl;
```

```
ACGTGCAT
```

Move conversion. If the source sequence is not needed any more after the conversion, it is always advisable to use move instead of assign. The function move does not make a copy but can reuse the source sequence storage. In some cases, move can also perform an in-place conversion.

```
String<char> char_source = "acgtgcat";
String<Dna> dna_target;

// The in-place move conversion.
move(dna_target, char_source);
std::cout << dna_target << std::endl;
```

```
ACGTGCAT
```

Assignment 3

Type Review

Objective In this assignment you will sort nucleotides. Copy the code below. Adjust the code such that all nucleotides, which are lexicographically smaller than a Dna5 `'G'` are stored in a list `lesser`, while all nucleotides which are greater, should be stored in a list `greater`. Print out the final lists.

```
#include <seqan/stream.h>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    String<Dna5> nucleotides = "AGTCGTGNNANCT";
    String<Dna5> selected;
    // Append all elements of nucleotides, apart of Gs,
    // to the list selected.
    for (unsigned i = 0; i < length(nucleotides); ++i)
    {
        appendValue(selected, nucleotides[i]);
    }
    std::cout << "Selected nucleotides: " << selected << std::endl;
    return 0;
}
```

Solution Click **more...** to see the solution.

```
#include <seqan/stream.h>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    String<Dna5> nucleotides = "AGTCGTGNNANCT";
    String<Dna5> lesser;
    String<Dna5> greater;

    for (unsigned i = 0; i < length(nucleotides); ++i){
        if (nucleotides[i] < 'G')
            appendValue(lesser, nucleotides[i]);
        else if (nucleotides[i] > 'G')
            appendValue(greater, nucleotides[i]);
    }
    std::cout << "Lesser nucleotides: " << lesser << std::endl;
    std::cout << "Greater nucleotides: " << greater << std::endl;
}
```

```
Lesser nucleotides: ACAC
Greater nucleotides: TTNNNT
```

Assignment 4

Type Transfer

Objective In this task you will compare whole sequences. Reuse the code from above. Instead of a `String<Dna5>` we will now deal with a `String<Dna5String>`. Build a string which contains the Dna5Strings

"ATATANGCGT", "AAGCATGANT" and "TGAAANTGAC". Now check for all elements of the container, if they are lexicographically smaller or bigger than the given subject sequence "GATGCATGAT" and append them to a appropriate list. Print out the final lists.

Hints Try to avoid unnecessary sequence scans.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

int main()
{
    String<Dna5String> nucleotidesList;
    Dna5String str1 = "ATATANGCGT";
    Dna5String str2 = "AAGCATGANT";
    Dna5String str3 = "TGAAANTGAC";
    resize(nucleotidesList, 3);
    nucleotidesList[0] = str1;
    nucleotidesList[1] = str2;
    nucleotidesList[2] = str3;

    String<Dna5String> lesser;
    String<Dna5String> greater;
    Dna5String ref = "GATGCATGAT";

    // For each Dna5String of the String:
    for (unsigned i = 0; i < length(nucleotidesList); ++i)
    {
        // Compare the Dna5String with the given reference string
        // The result of the comparison is stored in comp
        Lexical<> comp(nucleotidesList[i], ref);
        // The function isLess checks only the stored result
        // without comparing the sequences again
        if (isLess(comp))
            appendValue(lesser, nucleotidesList[i]);
        else if (isGreater(comp))
            appendValue(greater, nucleotidesList[i]);
    }
    // Print the results
    std::cout << "Lesser sequences: " << std::endl;
    for (unsigned i = 0; i < length(lesser); ++i)
    {
        std::cout << lesser[i] << ", ";
    }
    std::cout << std::endl;
    std::cout << "Greater sequences: " << std::endl;
    for (unsigned i = 0; i < length(greater); ++i)
    {
        std::cout << greater[i] << ", ";
    }
}
```

```
Lesser sequences:
ATATANGCGT, AAGCATGANT,
Greater sequences:
```

```
TGAAANTGAC,
```

Iteration

Very often you will be required to iterate over your string to either retrieve what's stored in the string or to write something at a specific position. For this purpose SeqAn provides Iterators for all container types. The metafunction Iterator can be used to determine the appropriate iterator type for a given a container.

An iterator always points to one value of the container. The operator operator* can be used to access this value by reference. Functions like operator++(prefix) or operator–(prefix) can be used to move the iterator to other values within the container.

The functions begin and end, applied to a container, return iterators to the begin and to the end of the container. Note that similar to C++ standard library iterators, the iterator returned by end does not point to the last value of the container but to the position behind the last one. If the container is empty then `end() == begin()`.

The following code prints out a sequence and demonstrates how to iterate over a string.

```
DnaString genome = "ACGTACGTACGT";
typedef Iterator<DnaString>::Type TIterator;
for (TIterator it = begin(genome); it != end(genome); ++it)
{
    std::cout << *it;
}
```

```
ACGTACGTACGT
```

Different Iterator Types

Some containers offer several kinds of iterators, which can be selected by an optional template parameter of the Iterator class. For example, the tag Standard can be used to get an iterator type that resembles the C++ standard random access iterator. For containers there is also a second variant available, the so called Rooted iterator. The rooted iterator knows its container by pointing back to it. This gives us a nice interface to access member functions of the underlying container while operating on a rooted iterator. The construction of an iterator in SeqAn, e.g. for a Dna String, could look like the following:

```
Iterator<DnaString>::Type           it1;  // A standard iterator
Iterator<DnaString, Standard>::Type it2;  // Same as above
Iterator<DnaString, Rooted>::Type   it3;  // A rooted iterator
```

Tip: The default iterator implementation is Standard. Rooted iterators offer some convenience interfaces for the user. They offer additional functions like container for determining the container on which the iterator works, and they simplify the interface for other functions like atEnd. Moreover, rooted iterators may change the container's length or capacity, which makes it possible to implement a more intuitive variant of a remove algorithm.

While rooted iterators can usually be converted into standard iterators, it is not always possible to convert standard iterators back into rooted iterators, since standard iterators may lack the information about the container they work on. Therefore, many functions that return iterators like begin or end return rooted iterators instead of standard iterators; this way, they can be used to set both rooted and standard iterator variables. Alternatively it is possible to specify the returned iterator type explicitly by passing the iterator kind as a tag argument, e.g. `begin(str, Standard())`.

Assignment 5

Type Review

Objective

Copy the code below, which replaces all N's of a given String with A's. Adjust the code to use iterators to traverse the container. Use the Standard iterator.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

int main()
{
    Dna5String genome = "ANTGGTTNCAACNGTAANTGCTGANNNACATGTNCGCGTGTA";
    for (unsigned i = 0; i < length(genome); ++i)
    {
        if (genome[i] == 'N')
            genome[i] = 'A';
    }
    std::cout << "Modified genome: " << genome << std::endl;
    return 0;
}
```

Solution

Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

int main()
{
    Dna5String genome = "ANTGGTTNCAACNGTAANTGCTGANNNACATGTNCGCGTGTA";

    Iterator<Dna5String>::Type it = begin(genome);
    Iterator<Dna5String>::Type itEnd = end(genome);

    for (; it != itEnd; goNext(it))
    {
        if (getValue(it) == 'N')
            value(it) = 'A';
    }
    std::cout << "Modified genome: " << genome << std::endl;
    return 0;
}
```

Assignment 6

Type Application

Objective Use the code from above and change the Standard to a Rooted iterator. Try to shorten the code wherever possible.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

int main()
{
    Dna5String genome = "ANTGGTTNCAACNGTAANTGCTGANNNACATGTNCGCGTGTA";

    Iterator<Dna5String, Rooted>::Type it = begin(genome);

    for (; !atEnd(it); goNext(it))
    {
        if (getValue(it) == 'N')
            value(it) = 'A';
    }
    std::cout << "Modified genome: " << genome << std::endl;
    return 0;
}
```

String Allocation Strategies

Each sequence object has a capacity, i.e. the reserved space for this object. The capacity can be set explicitly by functions such as reserve or resize. It can also bet set implicitly by functions like append, assign, insert or replace, if the operation's result exceeds the length of the target sequence.

If the current capacity of a sequence is exceeded by chaining the length, we say that the sequence overflows. There are several overflow strategies that determine what actually happens when a string should be expanded beyond its capacity. The user can specify this for a function call by additionally handing over a tag. If no overflow strategy is specified, a default overflow strategy is selected depending on the type of the sequence.

The following overflow strategies exist:

Exact Expand the sequence exactly as far as needed. The capacity is only changed if the current capacity is not large enough.

Generous Whenever the capacity is exceeded, the new capacity is chosen somewhat larger than currently needed. This way, the number of capacity changes is limited in a way that resizing the sequence only takes amortized constant time.

Limit Instead of changing the capacity, the contents are limited to current capacity. All values that exceed the capacity are lost.

Insist No capacity check is performed, so the user has to ensure that the container's capacity is large enough.

The next example illustrates how the different strategies could be used:

```
String<Dna> dnaSeq;
// Sets the capacity of dnaSeq to 5.
resize(dnaSeq, 4, Exact());
// Only "TATA" is assigned to dnaSeq, since dnaSeq is limited to 4.
assign(dnaSeq, "TATAGGGG", Limit());
std::cout << dnaSeq << std::endl;
```

```
    // Use the default expansion strategy.
    append(dnaSeq, "GCGCGC");
    std::cout << dnaSeq << std::endl;
```

```
TATA
TATAGCGCGC
```

Assignment 7

Type Review

Objective Build a string of Dna (default specialization) and use the function `appendValue` to append a million times the nucleotide 'A'. Do it both using the overflow strategy `Exact` and `Generous`. Measure the time for the two different strategies.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>

using namespace seqan;

int main()
{
    unsigned num = 100000;
    double start;

    String<Dna> str;
    clear(str);
    start = sysTime();
    for (unsigned i = 0; i < num; ++i)
        appendValue(str, 'A', Exact());
    std::cout << "Strategy Exact() took: " << sysTime() - start << " s\n\n";

    clear(str);
    shrinkToFit(str);
    start = sysTime();
    for (unsigned i = 0; i < num; ++i)
        appendValue(str, 'A', Generous());
    std::cout << "Strategy Generous() took: " << sysTime() - start << " s\n\n";

    return 0;
}
```

String Specializations

The user can specify the kind of string that should be used in an optional second template argument of String. The default string implementation is Alloc String.

```

```

In most cases, the implementation Alloc String (the default when using a `String<T>`) is the best choice. Exceptions are when you want to process extremely large strings that are a bit larger than the available memory (consider Alloc

String) or much larger so most of them are stored on the hard disk and only parts of them are loaded in main memory (consider External String). The following list describes in detail the different specializations:

Specialization Alloc String

- **Description** Expandable string that is stored on the heap.
- **Applications** The default string implementation that can be used for general purposes.
- **Limitations** Changing the capacity can be very costly since all values must be copied.

Specialization Array String

- **Description** Fast but non-expandable string. Fast storing of fixed-size sequences.
- **Limitations** Capacity must already be known at compile time. Not suitable for storing large sequences.

Specialization Block String

- **Description** String that stores its sequence characters in blocks.
- **Applications** The capacity of the string can quickly be increased. Good choice for growing strings or stacks.
- **Limitations** Iteration and random access to values is slightly slower than for Alloc String.

Specialization Packed String

- **Description** A string that stores as many values in one machine word as possible.
- **Applications** Suitable for storing large strings in memory.
- **Limitations** Slower than other in-memory strings.

Specialization External String

- **Description** String that is stored in secondary memory.
- **Applications** Suitable for storing very large strings (>2GB). Parts of the string are automatically loaded from secondary memory on demand.
- **LimitationsApplications** Slower than other string classes.

Specialization Journaled String

- **Description** String that stores differences to an underlying text rather than applying them directly.
- **Applications** Suitable for efficiently storing similar strings, if their differences to an underlying reference sequence are known.
- **LimitationsApplications** Slower than other string classes, due to logarithmic penalty for random accesses.

Specialization CStyle String

- **Description** Allows adaption of strings to C-style strings.
- **Applications** Used for transforming other String classes into C-style strings (i.e. null terminated char arrays). Useful for calling functions of C-libraries.
- **Limitations** Only sensible if value type is `char` or `wchar_t`.

```
// String with maximum length 100.
String<char, Array<100> > myArrayString;
// String that takes only 2 bits per nucleotide.
String<Dna, Packed<> > myPackedString;
```

```
// Most of the string is stored on the disk.
String<Dna, External<> > myLargeGenome;
```

Tip: String Simplify Memory Management

One advantage of using Strings is that the user does not need to reserve memory manually with **new** and does not need **delete** to free memory. Instead, those operations are automatically handled by the String class.

```
    String<Dna> myDnaGenome = "TATACGCG";
```

Segments

The following section will introduce you into the Segment class of SeqAn.

Segments are contiguous subsequences that represent parts of other sequences. Therefore, their functionality is similar to the String functionality. In SeqAn, there are three kinds of segments: InfixSegment, PrefixSegment, and SuffixSegment. The metafunctions Infix, Prefix, and Suffix, respectively, return the appropriate segment data type for a given sequence type.

For prefixes, we use the function prefix to build the prefix. The first parameter is the sequence we build the prefix from, the second the **excluding** end position. For infixes, we have to provide both the including start and the excluding end position. For suffixes, the second parameter of the function denotes the including starting position of the suffix:

```
    String<Dna> dnaSeq = "AGTTGGCATG";
    Prefix<String<Dna> >::Type pre = prefix(dnaSeq, 4);
    std::cout << "Prefix: " << pre << std::endl;

    Infix<String<Dna> >::Type inf = infix(dnaSeq, 4, 7);
    std::cout << "Infix: " << inf << std::endl;

    Suffix<String<Dna> >::Type suf = suffix(dnaSeq, 4);
    std::cout << "Suffix: " << suf << std::endl;
```

```
Prefix: AGTT
Infix: GGC
Suffix: GGCATG
```

Segments store a pointer on the underlying sequence object, the *host*, and an start and/or end position, depending on the type of segment. The segment is *not* a copy of the sequence segment.

Warning: Please note that it is not possible anymore to change the underlying sequence by changing the segment. If you want to change the host sequence, you have to explicitly modify this. If you want to modify only the segment, you have to explicitly make a copy of the string.

Assignment 8

Type Application

Objective In this task you will use a segment to pass over an infix of a given sequence to a function without copying the corresponding fragment. Use the code given below. Lets assume that we have given a `genome` and a `read` sequence as well as the begin position of a given alignment. In the main function a fragment of the Dna5String `genome` is copied and passed together with the Dna5String `read` to a `print` function. Adjust the code to use an infix of the genome, instead of copying the corresponding fragment.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

// Function to print simple alignment between two sequences with the same length
// .. for two sequences of different types
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout <<  "Alignment " << std::endl;
    std::cout << "  genome : ";
    std::cout << genomeFragment << std::endl;
    std::cout << "  read   : ";
    std::cout << read << std::endl;
}

int main()
{
    // We have given a genome sequence
    Dna5String genome = "ATGGTTTCAACGTAATGCTGAACATGTCGCGT";
    // A read sequence
    Dna5String read = "TGGTNTCA";
    // And the begin position of a given alignment between the read and the genome
    unsigned beginPosition = 1;
```

```
    // We have to create a copy of the corresponding fragment of the genome,␣
↪where the read aligns to
    // Change this piece of code using an infix of the genome
    for (unsigned i = 0; i < length(read); ++i)
    {
        appendValue(genomeFragment, genome[beginPosition + i]);
    }
```

```
    // Call of our function to print the simple alignment
    printAlign(genomeFragment, read);
    return 0;
}
```

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

// Function to print simple alignment between two sequences with the same length
// .. for two sequences of different types
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout <<  "Alignment " << std::endl;
    std::cout << "  genome : ";
    std::cout << genomeFragment << std::endl;
    std::cout << "  read   : ";
```

```
    std::cout << read << std::endl;
}

int main()
{
    // We have given a genome sequence
    Dna5String genome = "ATGGTTTCAACGTAATGCTGAACATGTCGCGT";
    // A read sequence
    Dna5String read = "TGGTNTCA";
    // And the begin position of a given alignment between the read and the genome
    unsigned beginPosition = 1;

    // Create Infix of type Dna5String and get the corresponding infix sequence␣
↪of genome
    Infix<Dna5String>::Type genomeFragment = infix(genome, beginPosition,␣
↪beginPosition + length(read));

    // Call of our function to print the simple alignment
    printAlign(genomeFragment, read);
    return 0;
}
```

```
Alignment
  genome : TGGTTTCA
  read   : TGGTNTCA
```

Assignment 9

Type Review

Objective Take the solution from the workshop assignment above and change it to use Segments for building the genome fragment.

Hints Note that because `printAlign` uses templates, you don't have to change the function even though the type of `genomeFragment` is different.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;
// Function to print simple alignment between two sequences with the same length
template <typename TText1, typename TText2>
void printAlign(TText1 const & genomeFragment, TText2 const & read)
{
    std::cout <<  "Alignment " << std::endl;
    std::cout << "  genome : " << genomeFragment << std::endl;
    std::cout << "  read   : " << read << std::endl;
}

int main(int, char const **)
{
    // Build reads and genomes
    DnaString chr1 =
↪"TATAATATTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCTAGATGTGCAGCTCGATCGAATGCACGTG
↪";
```

```
    // Build List containing all reads
    typedef String<DnaString> TDnaList;
    TDnaList readList;
    resize(readList, 4);
    readList[0] = "TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT";
    readList[1] =
↪"TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT";
    readList[2] =
↪"AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGGTTCATGTGCGCTGAAGCACACATGCACA";
    readList[3] = "CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA";
    // Append a second chromosome sequence fragment to chr1
    DnaString chr2 =
↪"AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGGTTCATGTGCGCTGAAGCACACATGCACACGTCTCTGTGTTCCGACG
↪";
    append(chr1, chr2);
    // Print readlist
    std::cout << " \n Read list: " << std::endl;
    for (unsigned i = 0; i < length(readList); ++i)
        std::cout << readList[i] << std::endl;
    // Assume we have mapped the 4 reads to chr1 (and chr2) and now have the␣
↪mapping start positions (no gaps).
    // Store the start position in a String alignPosList: 7, 100, 172, 272
    String<unsigned> alignPosList;
    resize(alignPosList, 4);
    alignPosList[0] = 7;
    alignPosList[1] = 100;
    alignPosList[2] = 172;
    alignPosList[3] = 272;
    // Optional
    // Bisulfite conversion
    // Assume chr1 is beeing bisulfate treated: Copy chr1 to a new genome bsChr1␣
↪and exchange every 'C' with a 'T'
    DnaString bsChr1;
    assign(bsChr1, chr1);
    for (unsigned i = 0; i < length(bsChr1); ++i)
        if (bsChr1[i] == 'C')
            bsChr1[i] = 'T';
    // Print alignments using Segment: Do the same as above, but instead of using␣
↪a for loop to build the fragment,
    // use the Segment class to build an infix of bsChr1.
    // Note: Because printAlign uses templates, we don't have to change the␣
↪function even though the type of
    // genomeFragment is different.
    std::cout << " \n Print alignment using Segment: " << std::endl;
    for (unsigned i = 0; i < length(readList); ++i)
    {
        // Begin and end position of a given alignment between the read and the␣
↪genome
        unsigned beginPosition = alignPosList[i];
        unsigned endPosition = beginPosition + length(readList[i]);
        // Build infix
        Infix<DnaString>::Type genomeFragment = infix(chr1, beginPosition,␣
↪endPosition);
        // Call of our function to print the simple alignment
        printAlign(genomeFragment, readList[i]);
    }
    return 0;
}
```

```
 Read list:
TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT
TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT
AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACA
CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA

 Print alignment using Segment:
Alignment
  genome : TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT
  read   : TTGCTATCGCGATATCGCTAGCTAGCTACGGATTATGCGCTCTGCGATATATCGCGCT
Alignment
  genome : TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT
  read   : TCGATTAGCGTCGATCATCGATCTATATTAGCGCGCGGTATCGGACGATCATATTAGCGGTCTAGCATT
Alignment
  genome : ␣
↪AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACA
  read   : ␣
↪AGCCTGCGTACGTTGCAGTGCGTGCGTAGACTGTTGCAAGCCGGGGGTTCATGTGCGCTGAAGCACACATGCACA
Alignment
  genome : CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA
  read   : CGTGCACTGCTGACGTCGTGGTTGTCACATCGTCGTGCGTGCGTACTGCTGCTGACA
```

ToC

Contents

String Sets

Learning Objective You will learn the advantages of StringSets and how to work with them.

Difficulty Basic

Duration 15 min

Prerequisites *Sequences*

A set of sequences can either be stored in a sequence of sequences, for example in a `String<String<char> >`, or in a StringSet. This tutorial will introduce you to the SeqAn class StringSet, its background and how to use it.

Background

One advantage of using StringSet is that it supports the function concat that returns a *concatenator* of all sequences in the string set. A *concatenator* is an object that represents the concatenation of a set of strings. This way, it is

possible to build up index data structures for multiple sequences by using the same construction methods as for single sequences.

There are two kinds of StringSet specializations in SeqAn: Owner StringSet, the default specialisation, and Dependent StringSet; see the list below for details. Owner StringSets actually store the sequences, whereas Dependent StringSets just refer to sequences that are stored outside of the string set.

```
StringSet<DnaString>                ownerSet;
StringSet<DnaString, Owner<> >      ownerSet2;      // same as above
StringSet<DnaString, Dependent<> > dependentSet;
```

The specialization ConcatDirect StringSet already stores the sequences in a concatenation. The concatenators for all other specializations of StringSet are **virtual** sequences, that means their interface **simulates** a concatenation of the sequences, but they do not literally concatenate the sequences into a single sequence. Hence, the sequences do not need to be copied when a concatenator is created.

One string can be an element of several Dependent StringSets. Typical tasks are, e.g., to find a specific string in a string set, or to test whether the strings in two string sets are the same. Therefore a mechanism to identify the strings in the string set is needed, and, for performance reasons, this identification should not involve string comparisons. SeqAn solves this problem by introducing *ids*, which are by default `unsigned int` values.

The following list lists the different StringSet specializations:

Specialization `Owner<ConcatDirect>` The sequences are stored as parts of a long string. Since the sequences are already concatenated, concat just needs to return this string. The string set also stores lengths and starting positions of the strings. Inserting new strings into the set or removing strings from the set is more expensive than for the default OwnerStringSet specialization, since this involves moving all subsequent sequences in memory.

Specialization `Owner<JournaledSet>` The sequences are stored as Journaled Strings to a common reference sequence, that is also stored within the container. When adding a new String to the set, it needs to be joined to this set of sequences which are all based on the common reference sequence. This way one can hold a large collection of similar sequences efficiently in memory.

Specialization `Dependent<Tight>` This specialization stores sequence pointers consecutively in an array. Another array stores an id value for each sequence. That means that accessing given an id needs a search through the id array.

> **Warning:** The Dependent-Tight StringSet is deprecated and will likely be removed within the SeqAn-2.x lifecycle.

Specialization `Dependent<Generous>` The sequence pointers are stored in an array at the position of their ids. If a specific id is not present, the array stores a zero at this position. The advantage of this specialization is that accessing the sequence given its id is very fast. On the other hand, accessing a sequence given its position `i` can be expensive, since this means we have to find the *i*-th non-zero value in the array of sequence pointers. The space requirements of a string set object depends on the largest id rather than the number of sequences stored in the set. This could be inefficient for string sets that store a small subset out of a large number of sequences.

Building String Sets

Use the function appendValue to append strings to string sets.

```
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;
```

```
int main()
{
    StringSet<DnaString> stringSet;
    DnaString str0 = "TATA";
    DnaString str1 = "CGCG";
    appendValue(stringSet, str0);
    appendValue(stringSet, str1);
```

Working with StringSets

This section will give you a short overview of the functionality of the class StringSet.

There are two ways for accessing the sequences in a string set: (1) the function operator[] returns a reference to the sequence at a specific *position* within the sequence of sequences, and (2) valueById accesses a sequence given its *id*. We can retrieve the *id* of a sequence in a StringSet with the function positionToId.

```
    // (1) Access by position
    std::cout << "Owner: " << '\n';
    std::cout << "Position 0: " << value(stringSet, 0) << '\n';

    // Get the corresponding ids
    unsigned id0 = positionToId(stringSet, 0);
    unsigned id1 = positionToId(stringSet, 1);

    // (2) Access by id
    std::cout << "Id 0:  " << valueById(stringSet, id0) << '\n';
    std::cout << "Id 1:  " << valueById(stringSet, id1) << '\n';

    return 0;
}
```

```
Owner:
Position 0: TATA
Id 0:  TATA
Id 1:  CGCG
```

In the case of Owner StringSets, id and position of a string are always the same, but for Dependent StringSets, the ids can differ from the positions. For example, if a Dependent StringSet is used to represent subsets of strings that are stored in Owner StringSets, one can use the position of the string within the Owner StringSet as id of the strings. With the function assignValueById, we can add the string with a given id from the source string set to the target string set.

```
    // Let's create a string set of type dependent to represent strings,
    // which are stored in the StringSet of type Owner
    StringSet<DnaString, Dependent<Tight> > depSet;
    // We assign the first two strings of the owner string set to the dependent␣
↪StringSet,
    // but in a reverse order
    assignValueById(depSet, stringSet, id1);
    assignValueById(depSet, stringSet, id0);

    std::cout << "Dependent: " << '\n';
    // (1) Access by position
    std::cout << "Pos 0: " << value(depSet, 0) << '\n';
    // (2) Access by id
    std::cout << "Id 0:  " << valueById(depSet, id0) << '\n';
```

```
Dependent:
Pos 0: CGCG
Id 0:  TATA
```

With the function positionToId we can show that, in this case, the position and the id of a string are different.

```
    std::cout << "Position 0: Id " << positionToId(depSet, 0) << '\n';
    std::cout << "Position 1: Id " << positionToId(depSet, 1) << '\n';
```

```
Position 0: Id 1
Position 1: Id 0
```

Iterating over String Sets

As well as for other containers, SeqAn has implemented iterators for StringSets. The following example illustrates, how to iterate over the StringSet.

```
    typedef Iterator<StringSet<DnaString> >::Type TStringSetIterator;
    for (TStringSetIterator it = begin(stringSet); it != end(stringSet); ++it)
    {
        std::cout << *it << '\n';
    }
```

```
TATA
CGCG
```

If we want to iterate over the contained Strings as well, as if the StringSet would be one sequence, we can use the function concat to get the concatenation of all sequences. Therefore we first use the metafunction Concatenator to receive the type of the concatenation. Then, we can simply build an iterator for this type and iterate over the concatenation of all strings.

```
    typedef Concatenator<StringSet<DnaString> >::Type TConcat;
    TConcat concatSet = concat(stringSet);

    Iterator<TConcat>::Type it = begin(concatSet);
    Iterator<TConcat>::Type itEnd = end(concatSet);
    for (; it != itEnd; goNext(it))
    {
        std::cout << getValue(it) << " ";
    }
    std::cout << '\n';
```

```
T A T A C G C G
```

Assignment 1

Type Review

Objective Build a string set with default specialization and which contains the strings `"AAA"`, `"CCC"`, `"GGG"` and `"TTT"`. After that print the length of the string set and use a simple for-loop to print all elements of the strings set.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

int main()
{
    // Build strings
    DnaString str0 = "AAA";
    DnaString str1 = "CCC";
    DnaString str2 = "GGG";
    DnaString str3 = "TTT";
    // Build string set and append strings
    StringSet<DnaString> stringSet;
    appendValue(stringSet, str0);
    appendValue(stringSet, str1);
    appendValue(stringSet, str2);
    appendValue(stringSet, str3);
    // Print the length of the string set
    std::cout << length(stringSet) << std::endl;
    // Print all elements
    for (unsigned i = 0; i < length(stringSet); ++i)
    {
        std::cout << stringSet[i] << std::endl;
    }
    return 0;
}
```

Assignment 2

Type Application

Objective In this task you will test, whether a Dependent StringSet contains a string without comparing the actual sequences. Use the given code frame below and adjust it in the following way:

1. Build a Owner StringSet to store the given strings.
2. Get the corresponding ids for each position and store them.
3. Build a DependentStringSet and assign the strings of the owner string set from position 0,1 and 3 by their id to it.
4. Write a function `isElement` which takes a `StringSet<Dependent<> >` and a `Id` as arguments and checks whether a string set contains a string with a given id.
5. Check if the string set contains the string of position `3` and `2` and print the result.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/file.h>

using namespace seqan;

int main()
{
    // Build strings
```

```
    DnaString str0 = "TATA";
    DnaString str1 = "CGCG";
    DnaString str2 = "TTAAGGCC";
    DnaString str3 = "ATGC";
    DnaString str4 = "AGTGTCA";

    // Your code

    return 0;
}
```

Hints You can use the SeqAn functions positionToId and assignValueById.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

// Check whether the string set contains the string with the given id,
// without comparing the actual sequences
template <typename TStringSet, typename TId>
bool isElement(TStringSet & stringSet1, TId & id)
{

    for (unsigned i = 0; i < length(stringSet1); ++i)
    {
        // Get the id of the element at position i
        if (positionToId(stringSet1, i) == id)
            return true;
    }
    return false;
}

int main()
{
    // Build strings
    DnaString str0 = "TATA";
    DnaString str1 = "CGCG";
    DnaString str2 = "TTAAGGCC";
    DnaString str3 = "ATGC";
    DnaString str4 = "AGTGTCA";
    // Build owner string set and append strings
    StringSet<DnaString> stringSetOw;
    appendValue(stringSetOw, str0);
    appendValue(stringSetOw, str1);
    appendValue(stringSetOw, str2);
    appendValue(stringSetOw, str3);
    appendValue(stringSetOw, str4);
    // Get corresponding ids for positions
    unsigned id0 = positionToId(stringSetOw, 0);
    unsigned id1 = positionToId(stringSetOw, 1);
    unsigned id2 = positionToId(stringSetOw, 2);
    unsigned id3 = positionToId(stringSetOw, 3);
    // Build dependent string set and assigns strings by id
    StringSet<DnaString, Dependent<Generous> > stringSetDep;
```

```
    assignValueById(stringSetDep, stringSetOw, id0);
    assignValueById(stringSetDep, stringSetOw, id1);
    assignValueById(stringSetDep, stringSetOw, id3);
    // Call function to check if a string is contained and output result
    std::cout << "Does the string set contain the string with the id 'id3'? " <<
↪isElement(stringSetDep, id3) << std::endl;
    std::cout << "Does the string set contain the string with the id 'id2'? " <<
↪isElement(stringSetDep, id2) << std::endl;

    return 0;
}
```

ToC

Contents

Alphabets

Learning Objective You will learn the details about the alphabets in SeqAn.

Difficulty Basic

Duration 15 min

Prerequisites *A First Example*

This tutorial will describe the different alphabets used in SeqAn, or in other words, you will learn about the contained types of a SeqAn String. To continue with the other tutorials, it would be enough to know, that in SeqAn several standard alphabets are already predefined, e.g. Dna, Dna5, Rna, Rna5, Iupac, AminoAcid.

Types

Any type that provides a default constructor, a copy constructor and an assignment operator can be used as the alphabet / contained type of a String (see also the tutorial *Sequences*). This includes the C++ POD types, e.g. `char`, `int`, `double` etc. In addition you can use more complex types like String as the contained type of strings, e.g. `String<String<char> >`.

SeqAn also provides the following types that are useful in bioinformatics. Each of them is a specialization of the class SimpleType.

Specialization	Description
AminoAcid	Amino Acid Alphabet
Dna	DNA alphabet
Dna5	N alphabet including N character
DnaQ	N alphabet plus phred quality
Dna5Q	N alphabet plus phred quality including N character
Finite	Finite alphabet of fixed size.
Iupac	N Iupac code.
Rna	N alphabet
Rna5	N alphabet including N character

Functionality

In SeqAn, alphabets are value types that can take a limited number of values and which hence can be mapped to a range of natural numbers. We can retrieve the number of different values of an alphabet, the alphabet size, by the metafunction ValueSize.

```
    typedef Dna TAlphabet;

    unsigned alphSize = ValueSize<TAlphabet>::VALUE;
    std::cout << "Alphabet size of Dna: " << alphSize << '\n';
```

```
Alphabet size of Dna: 4
```

Another useful metafunction called BitsPerValue can be used to determine the number of bits needed to store a value of a given alphabet.

```
    unsigned bits = BitsPerValue<TAlphabet>::VALUE;
    std::cout << "Number of bits needed to store a value of type Dna: " << bits << '\n
↪';
```

```
Number of bits needed to store a value of type Dna: 2
```

The order of a character in the alphabet (i.e. its corresponding natural number) can be retrieved by calling the function ordValue. See each specialization's documentation for the ordering of the alphabet's values.

```
    Dna a = 'A';
    Dna c = 'C';
    Dna g = 'G';
    Dna t = 'T';

    std::cout <<"A: " << (unsigned)ordValue(a) << '\n';
    std::cout <<"C: " << (unsigned)ordValue(c) << '\n';
    std::cout <<"G: " << (unsigned)ordValue(g) << '\n';
    std::cout <<"T: " << (unsigned)ordValue(t) << '\n';
```

```
A: 0
C: 1
G: 2
T: 3
```

Tip: The return value of the ordValue function is determined by the metafunction ValueSize. ValueSize returns the type which uses the least amount of memory while being able to represent all possible values. E.g. ValueSize of Dna

returns an `_uint8` which is able to represent 256 different characters. However, note that `std::cout` has no visible symbol for printing all values on the screen, hence a cast to `unsigned` might be necessary.

Assignment 1

Type Application

Objective In this task you will learn how to access all the letters of an alphabet. Use the piece of code from below and adjust the function `showAllLettersOfMyAlphabet()` to go through all the characters of the current alphabet and print them.

```
#include <seqan/sequence.h>
#include <seqan/basic.h>
#include <iostream>

using namespace seqan;

// We want to define a function, which takes
// the alphabet type as an argument
template <typename TAlphabet>
void showAllLettersOfMyAlphabet(TAlphabet const &)
{
    // ...
}

int main()
{
    showAllLettersOfMyAlphabet(AminoAcid());
    showAllLettersOfMyAlphabet(Dna());
    showAllLettersOfMyAlphabet(Dna5());
    return 0;
}
```

Hints You will need the Metafunction ValueSize.

Solution Click **more...** to see the solution.

```
#include <seqan/sequence.h>
#include <seqan/basic.h>
#include <iostream>

using namespace seqan;

// We define a function which takes
// the alphabet type as an argument
template <typename TAlphabet>
void showAllLettersOfMyAlphabet(TAlphabet const &)
{
    typedef typename ValueSize<TAlphabet>::Type TSize;
    // We need to determine the alphabet size
    // using the metafunction ValueSize
    TSize alphSize = ValueSize<TAlphabet>::VALUE;
    // We iterate over all characters of the alphabet
    // and output them
    for (TSize i = 0; i < alphSize; ++i)
        std::cout << static_cast<unsigned>(i) << ',' << TAlphabet(i) << "  ";
    std::cout << std::endl;
```

```
}

int main()
{
    showAllLettersOfMyAlphabet(AminoAcid());
    showAllLettersOfMyAlphabet(Dna());
    showAllLettersOfMyAlphabet(Dna5());
    return 0;
}
```

Tasks, such as computing an alignment, searching for patterns online or indexing a genome or protein database are required in many bioinformatics applications. For these tasks, SeqAn provides fundamental data structures to work efficiently with biological sequences. SeqAn implements special alphabets, such as DNA, RNA, AminoAcid, Iupac and more. The alphabets available in SeqAn can be reviewed *here*. You will also find some more information about using the alphabet types.

Besides the alphabets SeqAn also implements a string class. SeqAn strings are generic containers in which characters of any alphabet are stored continuously in memory. The default string class implementation is equivalent to the STL vector class. However, the memory mangement of the SeqAn string class is optimized for working with SeqAn's alphabets. Apart of the default string class implementation SeqAn provides many useful specializations of this class, which are very useful in the bioinformatics context. The tutorial about *Strings and Segments* gives you a more detailed overview over the string class and it's functionality.

Another generic container data structure used very often is the string set. The string set is a special container to represent a collection of strings, which in addition provides many helpful functions to make the work even easier and more efficient. The *StringSet tutorial* introduces you to this class and how to work with it.

Indices

ToC

Contents

Q-gram Index

Learning Objective You will know the features of the q-gram Index, how it can be used for searching and how to access the different fibres.

Difficulty Average

Duration 1 h

Prerequisites *Sequences*

The Q-gram Index

A q-gram index can be used to efficiently retrieve all occurrences of a certain q-gram in the text. It consists of various tables, called fibres (see *Accessing Index Fibres*), to retrieve q-gram positions, q-gram counts, etc. However, it has no support for suffix tree iterators. A q-gram index must be specialized with a Shape type. A Shape defines q, the number of characters in a q-gram and possibly gaps between these characters. There are different specializations of Shape available:

Specialization	Modifiable	Number of Gaps
UngappedShape	-	0
SimpleShape	+	0
OneGappedShape	+	0/1
GappedShape	-	any
GenericShape	+	any

- *- fixed at compile time*, *+ can be changed at runtime*

Each shape evaluates a gapped or ungapped sequence of q characters to a hash value by the Functions hash, hashNext, etc. For example, the shape `1101` represents a 3-gram with one gap of length 1. This shape overlayed with the Dna text `"GATTACA"` at the third position corresponds to `"TT-C"`. The function hash converts this 3-gram into $61 = (\mathbf{3} \cdot 4 + \mathbf{3}) \cdot 4 + 1$. 4 is the alphabet size in this example (see ValueSize).

With hash and hashNext, we can compute the hash values of arbitrary / adjacent q-grams and a loop that outputs the hash values of all overlapping ungapped 3-grams could look as follows:

```
DnaString text = "AAAACACAGTTTGA";
Shape<Dna, UngappedShape<3> > myShape;

std::cout << hash(myShape, begin(text)) << '\t';
for (unsigned i = 1; i < length(text) - length(myShape) + 1; ++i)
    std::cout << hashNext(myShape, begin(text) + i) << '\t';
```

Note that the shape not only stores the length and gaps of a q-gram shape but also stores the hash value returned by the last hash/hashNext call. This hash value can be retrieved by calling value on the shape. However, one drawback of the example loop above is that the first hash value must be computed with hash while the hash values of the following overlapping q-grams can more efficiently be computed by hashNext. This complicates the structure of algorithms that need to iterate all hash values, as they have to handle this first hash differently. As a remedy, the hashInit function can be used first and then hashNext on the first and all following text positions in the same way:

```
hashInit(myShape, begin(text));
for (unsigned i = 0; i < length(text) - length(myShape) + 1; ++i)
    std::cout << hashNext(myShape, begin(text) + i) << '\t';
```

The q-gram index offers different functions to search or count occurrences of q-grams in an indexed text, see getOccurrences, countOccurrences. A q-gram index over a StringSet stores occurrence positions in the same way and in the same fibre (FibreSA) as the ESA index. If only the number of q-grams per sequence are needed the QGramCounts and QGramCountsDir fibres can be used. They store pairs `(seqNo, count)`, `count`>0, for each q-gram that occurs `counts` times in sequence number `seqNo`.

To efficiently retrieve all occurrence positions or all pairs `(seqNo, count)` for a given q-gram, these positions or pairs are stored in contiguous blocks (in QGramSA, QGramCounts fibres), called buckets. The begin position of bucket i is stored in directory fibres (QGramDir, QGramCountsDir) at position i, the end position is the begin positions of the bucket i+1. The default implementation of the IndexQGram index maps q-gram hash values 1-to-1 to bucket numbers. For large q or large alphabets the Open Addressing QGram Index can be more appropriate as its directories are additionally bound by the text length. This is realized by a non-trivial mapping from q-gram hashes to bucket numbers that requires an additional fibre (QGramBucketMap).

For more details on q-gram index fibres see *Accessing Index Fibres* or QGram Index Fibres.

Example

We want to construct the q-gram index of the string `"CATGATTACATA"` and output the occurrences of the ungapped 3-gram `"CAT"`. As 3 is fixed at compile-time and the shape has no gaps we can use an UngappedShape which is the first template argument of IndexQGram, the second template argument of Index. Next we create the string `"CATGATTACATA"` and specialize the first index template argument with the type of this string. The string can be given to the index constructor.

```
int main()
{
    typedef Index<DnaString, IndexQGram<UngappedShape<3> > > TIndex;
    TIndex index("CATGATTACATA");
```

To get all occurrences of a q-gram, we first have to hash it with a shape of the same type as the index shape (we can even use the index shape returned by indexShape). The hash value returned by hash or hashNext is also stored in the shape and is used by the function getOccurrences to retrieve all occurrences of our 3-gram.

```
    hash(indexShape(index), "CAT");
    for (unsigned i = 0; i < length(getOccurrences(index, indexShape(index))); ++i)
        std::cout << getOccurrences(index, indexShape(index))[i] << std::endl;

    return 0;
}
```

Program output:

```
0
8
```

Assignment 1

Type Review

Objective Write a program that outputs all occurrences of the gapped q-gram "AT-A" in "CATGATTACATA".

Solution Before we can create a DnaString index of "CATGATTACATA", we have to choose an appropriate Shape. Because our shape `1101` is known at compile-time and contains only one gap we could choose OneGappedShape, GappedShape, or GenericShape (see the commented-out code). Although the GenericShape could be used for every possible shape, it is a good idea to choose a Shape with restrictions as its hash functions are more efficient in general.

```
int main()
{
    Index<DnaString, IndexQGram<OneGappedShape> > index("CATGATTACATA");
    stringToShape(indexShape(index), "1101");
```

Please note that the Shape object that corresponds to the IndexQGram index is empty initially and has to be set by stringToShape or resize. This initialization is not necessary for Shape that are defined at compile-time, i.e. UngappedShape and GappedShape. To search for "AT-A" we first have to hash it with the index shape or any other Shape with the same bitmap. The we can use getOccurrences to output all matches.

```
    hash(indexShape(index), "ATCA");
    for (unsigned i = 0; i < length(getOccurrences(index, indexShape(index)));␣
↪++i)
        std::cout << getOccurrences(index, indexShape(index))[i] << std::endl;
```

```
    return 0;
}
```

Tip: Instead of `length(getOccurrences(...))` we could have used countOccurrences. But beware that countOccurrences requires only the `QGram_Dir` fibre, whereas getOccurrences requires both `QGram_Dir` and `QGram_SA`, see *Accessing Index Fibres*. Because `QGram_SA` can be much more efficiently constructed during the construction of `QGram_Dir`, `QGram_Dir` would be constructed twice.

Program output:

```
1
4
```

Assignment 2

Type Review

Objective Create and output a matrix M where M(i,j) is the number of common ungapped 5-grams between sequence i and sequence j for 3 random Dna sequences, each not longer than 200 characters. Optional: Run the matrix calculation twice, once for an IndexQGram and once for an Open Addressing QGram Index and output the directory sizes (QGram_Dir, QGram_CountsDir fibre).

Hint A common q-gram that occurs a times in one and b times in the other sequence counts for $\min(a, b)$.

Solution For generating random numbers we use the std::mt19937. The random numbers returned by the random number engine are arbitrary `unsigned int` values which we downscale to values between 0 and 3 and convert into Dna characters. The 3 generated strings are of random length and appended to a StringSet. The main algorithm is encapsulated in a template function `qgramCounting` to easily switch between the two IndexQGram specializations.

```
int main()
{
    //  for the sake of reproducibility
    std::mt19937 rng;

    // create StringSet of 3 random sequences
    StringSet<DnaString> stringSet;
    reserve(stringSet, 3);
    for (int seqNo = 0; seqNo < 3; ++seqNo)
    {
        DnaString tmp;
        int len = rng() % 100 + 10;
        for (int i = 0; i < len; ++i)
            appendValue(tmp, Dna(rng() % 4));
        appendValue(stringSet, tmp);
        std::cout << ">Seq" << seqNo << std::endl << tmp << std::endl;
    }

    qgramCounting(stringSet, IndexQGram<UngappedShape<5> >());
    qgramCounting(stringSet, IndexQGram<UngappedShape<5>, OpenAddressing>());
    return 0;
}
```

The main function expects the StringSet and the Index specialization as a tag. First, we define lots of types we need to iterate and access the fibres directly. We then notify the index about the fibres we require. For storing

the common q-grams we use a 2-dimensional Matrix object whose lengths have to be set with `setLength` for each dimension. The matrix is initialized with zeros by resize.

```
template <typename TStringSet, typename TIndexSpec>
void qgramCounting(TStringSet & set, TIndexSpec)
{
    typedef Index<TStringSet, TIndexSpec> TIndex;
    typedef typename Fibre<TIndex, QGramCounts>::Type TCounts;
    typedef typename Fibre<TIndex, QGramCountsDir>::Type TCountsDir;
    typedef typename Value<TCountsDir>::Type TDirValue;
    typedef typename Iterator<TCounts, Standard>::Type TIterCounts;
    typedef typename Iterator<TCountsDir, Standard>::Type TIterCountsDir;

    TIndex index(set);
    indexRequire(index, QGramCounts());

    // initialize distance matrix
    int seqNum = countSequences(index);
    Matrix<int, 2> distMat;
    setLength(distMat, 0, seqNum);
    setLength(distMat, 1, seqNum);
    resize(distMat, 0);

    std::cout << std::endl << "Length of the CountsDir fibre: " <<␣
↪length(indexCountsDir(index)) << std::endl;
    TIterCountsDir itCountsDir = begin(indexCountsDir(index), Standard());
    TIterCountsDir itCountsDirEnd = end(indexCountsDir(index), Standard());
    TIterCounts itCountsBegin = begin(indexCounts(index), Standard());
```

The main part of the function iterates over the CountsDir fibre. Each entry in this directory represents a q-gram bucket, a contiguous interval in the Counts fibre storing for every sequence the q-gram occurs in the number of occurrences in pairs (seqNo,count). The interval begin of each bucket is stored in the directory and the interval end is the begin of the next bucket. So the inner loops iterate over all non-empty buckets and two pairs (seqNo1,count1) and (seqNo2,count2) indicate that seqNo1 and seqNo2 have a common q-gram. At the end the matrix can simply be output by shifting it to the `cout` stream.

```
    // for each bucket count common q-grams for each sequence pair
    TDirValue bucketBegin = *itCountsDir;
    for (++itCountsDir; itCountsDir != itCountsDirEnd; ++itCountsDir)
    {
        TDirValue bucketEnd = *itCountsDir;

        // q-gram must occur in at least 2 different sequences
        if (bucketBegin != bucketEnd)
        {
            TIterCounts itA = itCountsBegin + bucketBegin;
            TIterCounts itEnd = itCountsBegin + bucketEnd;
            for (; itA != itEnd; ++itA)
                for (TIterCounts itB = itA; itB != itEnd; ++itB)
                    distMat((*itA).i1, (*itB).i1) += _min((*itA).i2, (*itB).i2);
        }
        bucketBegin = bucketEnd;
    }

    std::cout << std::endl << "Common 5-mers for Seq_i, Seq_j" << std::endl;
    std::cout << distMat;
}
```

Please note that the open addressing q-gram index directories are smaller than the IndexQGram index directories.

Program output:

```
>Seq0
GGCATCCGTTCAGTATACGCCT
>Seq1
GGACATACCGGTCCTAAGTACACGTGGCAGGGATGGTCGAAGAACCCGC
>Seq2
TGCAAAGTTAGCGTACTAGTTAGTAACCGTGATACTAGCAAAATGAGTCTTCTCGTCGAAGTGGACGGGATGTGCTCACGGCCTTTTTT

Length of the CountsDir fibre: 1025

Common 5-mers for Seq_i, Seq_j
18         0         0
0         45         5
0         0         85

Length of the CountsDir fibre: 257

Common 5-mers for Seq_i, Seq_j
18         0         0
0         45         5
0         0         85
```

ToC

Contents

String Indices

Learning Objective You will get an overview of the different kinds of indices in SeqAn and how they are used.

Difficulty Average

Duration 1 h

Prerequisites *Sequences*

Indices in SeqAn

Indices in SeqAn are substring indices, meaning that they allow efficient pattern queries in strings or sets of strings. In contrast to, e.g., online-search algorithms that search through the text in $\mathcal{O}(n)$, substring indices find a pattern in sublinear time $o(n)$.

You can find the following indices in SeqAn:

IndexSa Suffix Array *[MM93]*

IndexEsa Extended Suffix Array *[AKO04]*

IndexWotd Lazy suffix tree *[GKS03]*

IndexDfi Deferred Frequency Index *[WS08]*

IndexQGram Q-gram index (see here)

FMIndex Full-text minute index *[FM01]*

Index Construction

We will now show how we can create the different indices in SeqAn before we show how they are used for pattern search.

All the mentioned indices belong to the generic Index class. A SeqAn index needs two pieces of information: the type of the String or StringSet to be indexed and the index specialization, such as IndexEsa or FMIndex.

The following code snippet creates an enhanced suffix array index of a string of type Dna5.

```
String<Dna5> genome = "ACGTACGTACGTN";
Index<String<Dna5>, IndexEsa<> > esaIndex(genome);
```

In contrast, the next code snipped creates a FM index over a set of amino acid sequences:

```
StringSet<String<AminoAcid> > protein;
appendValue(protein, "VXLAGZ");
appendValue(protein, "GKTVXL");
appendValue(protein, "XLZ");

Index<StringSet<String<AminoAcid> >, FMIndex<> > fmIndex(protein);
```

Assignment 1

Type Review

Objective Copy the code below and

1. change it to build an IndexEsa over a string of type Dna,
2. add an IndexEsa over a StringSet of Strings of type Dna.

```
#include <seqan/sequence.h>
#include <seqan/index.h>

using namespace seqan;

int main()
{
```

```
    String<char> text = "This is the first example";
    Index<String<char>, FMIndex<> > index(text);

    return 0;
}
```

Solution

```
#include <seqan/sequence.h>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    // One possible solution to the first sub assignment
    String<Dna> text = "ACGTTTGACAGCT";
    Index<String<Dna>, IndexEsa<> > index(text);

    // One possible solution to the second sub assignment
    StringSet<String<Dna> > stringSet;
    appendValue(stringSet, "ACGTCATCAT");
    appendValue(stringSet, "ACTTTG");
    appendValue(stringSet, "CACCCCCCTATTT");

    Index<StringSet<String<Dna> >, IndexEsa<> > indexSet(stringSet);

    return 0;
}
```

Assignment 2

Type Application

Objective Write a small program that prints the locations of all occurrences of `"TATAA"` in `"TTATTAAGCGTATAGCCCTATAAATATAA"`.

Hints Use the find function as the conditional instruction of a <tt>while</tt> loop.

Solution

```
#include <seqan/sequence.h>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    String<Dna5> genome = "TTATTAAGCGTATAGCCCTATAAATATAA";
    Index<String<Dna5>, IndexEsa<> > esaIndex(genome);
    Finder<Index<String<Dna5>, IndexEsa<> > > esaFinder(esaIndex);

    while (find(esaFinder, "TATAA"))
    {
        std::cout << position(esaFinder) << std::endl;
    }
```

```
    return 0;
}
```

You might have noticed that we only applied the FMIndex and IndexEsa in the examples. The reason for this is that even though everything stated so far is true for the other indices as well, IndexWotd and IndexDfi are more useful when used with iterators as explained in the tutorial *Index Iterators* and the IndexQGram uses Shapes which is also explained in another tutorial.

One last remark is necessary.

Important: If you search for two different patterns with the same Finder object, you have to call the clear function of the finder between the search for the two patterns. Otherwise the behavior is undefined.

Handling Multiple Sequences (StringSets)

The previous sections already described how an index of a set of strings can be instantiated. A character position of a StringSet can be one of the following:

1. A local position (default), i.e. a Pair (seqNo, seqOfs) where seqNo identifies the string within the StringSet and the seqOfs identifies the position within this string.
2. A global position, i.e. a single integer value between 0 and the sum of string lengths minus 1. This integer is the position in the gapless concatenation of all strings in the StringSet to a single string.

For indices, the meta-function SAValue determines, which position type (local or global) will be used for internal index tables (suffix array, q-gram array) and what type of position is returned by functions like position of a Finder. SAValue returns a Pair (local position) by default, but could be specialized to return an integer type (global position) for some applications. If you want to write algorithms for both variants you should use the functions posLocalize, posGlobalize, getSeqNo, and getSeqOffset.

Storing and Loading

Storing and loading an index can be done with:

```
    const char * tempFileName = SEQAN_TEMP_FILENAME();
    save(index, tempFileName);
```

or

```
    open(index, tempFileName);
```

If you have built your q-gram index with variable shapes (i.e. SimpleShape GenericShape), you have to keep in mind that q or the shape is not stored or loaded. This must be done manually and directly before or after loading with resize or stringToShape.

A newly instantiated index is initially empty. If you assign a text to be indexed, solely the text fibre is set. All other fibres are empty and created on demand. Normally, a full created index should be saved to disk. Therefore, you have to create the required fibres explicitly by hand.

```
    indexRequire(index, FibreSA());
```

For the IndexEsa index you could do:

```
    indexRequire(esaIndex, EsaSA());
    indexRequire(esaIndex, EsaLcp());
    indexRequire(esaIndex, EsaChildtab());  // for TopDown iterators
    indexRequire(esaIndex, EsaBwt());       // for (Super-)MaxRepeats iterators
```

Indexes based on external strings, e.g. `Index<String<Dna,External<> >,IndexEsa<> >` or `Index<String<Dna,MMap<> >,IndexEsa<> >` cannot be saved, as they are persistent implicitly. The first thing after instantiating such an index should be associating it to a file with:

```
    Index<String<Dna, External<> >, IndexEsa<> > external_index;
    open(external_index, tempFileName);
```

The file association implies that any change on the index, e.g. fibre construction, is synchronized to disk. When instantiating and associating the index the next time, the index contains its previous state and all yet to be constructed fibres.

Reducing the memory consumption

All Indices in SeqAn are capable of indexing Strings or StringSets of arbitrary sizes, i.e. up to 2^64 characters. This always comes at a cost in terms of memory consumption, as any Index has to represent 64 bit positions in the underlying text. However, in many practical instances, the text to be indexed is shorter, e.g. it does not exceed 4.29 billion (2^32) characters. In this case, one can reduce the memory consumption of an Index by changing its internal data types, with no drawback concerning running time.

SA Fibre

All Indices in SeqAn internally use the FibreSA, i.e. some sort of suffix array. For Strings, each suffix array entry consumes 64 bit of memory per default, where 32 bit would be sufficient if the text size is appropriate. In order to change the size type of the suffix array entry we simply have to overload the metafunction SAValue.

```
template<>
struct SAValue<String<Dna> >
{
    typedef unsigned Type;
};
```

If your text is a StringSet, then SAValue will return a Pair that can be overloaded in the same way.

```
template<>
struct SAValue<StringSet<String<Dna> > >
{
    typedef Pair<unsigned, unsigned> Type;
};
```

The first type of the pair is used as the type for the index of a string in the string set. So if you only have a few strings you could save even more memory like this.

```
typedef Pair<unsigned char, unsigned> Type;
```

FMIndex Fibres

The size of a generalized FMIndex depends also on the total number of characters in a StringSet (see lengthSum). This trait can be configured via the FMIndexConfig object.

```
typedef FMIndexConfig<void, unsigned> TConfig;
Index<StringSet<String<Dna> >, FMIndex<void, TConfig> > configIndex(set);
```

Other Index Fibres

See *Accessing Index Fibres* for more information.

ToC

Contents

Index Iterators

Learning Objective You will know the different kinds of index indices and how to use them for searching.

Difficulty Average

Duration 1.5 h

Prerequisites *Sequences*

Virtual String Tree Iterator

SeqAn provides a common interface, called the Virtual String Tree Iterator (VSTree Iterator), which lets you traverse the IndexSa, IndexEsa, IndexWotd and IndexDfi as a suffix tree (*Indices* definition), the IndexQGram as a suffix trie, and the FMIndex as a prefix trie. In the first part of this tutorial we will concentrate on the TopDown Iterator which is one of the two index iterator specializations (besides the BottomUp Iterator). The second part will then deal with the DFS.

Top-Down Iteration

For index based pattern search or algorithms traversing only the upper parts of the suffix tree the TopDown Iterator or TopDown History Iterator is the best solution. Both provide the functions goDown and goRight to go down to the first child node or go to the next sibling. The TopDown History Iterator additionally provides goUp to go back to the parent node. The child nodes in IndexEsa indices are lexicographically sorted from first to last. For IndexWotd and IndexDfi indices this holds for all children except the first.

In the next example we want to use the TopDown Iterator to efficiently search a text for exact matches of a pattern. We therefore want to use goDown which has an overload to go down an edge beginning with a specific character.

Important: The following examples show how to iterate IndexSa, IndexEsa, IndexWotd or IndexDfi, i.e. Index specializations representing suffix trees. The result of the iteration will look different on Index specializations representing tries, e.g. FMIndex or IndexQGram. Indeed, the topology of an Index changes depending on the chosen tree or trie specialization. Note that any suffix tree edge can be labeled by more than one character, whereas any trie edge is always labeled by exactly one character.

First we create an index of the text `"How much wood would a woodchuck chuck?"`

```
int main()
{
    typedef Index<CharString> TIndex;
    TIndex index("How much wood would a woodchuck chuck?");

    Iterator<TIndex, TopDown<> >::Type it(index);

    CharString pattern = "wood";
    while (repLength(it) < length(pattern))
    {
        // go down edge starting with the next pattern character
        if (!goDown(it, pattern[repLength(it)]))
            return 0;

        unsigned endPos = std::min((unsigned)repLength(it),␣
↪(unsigned)length(pattern));
        // compare remaining edge characters with pattern
        std::cout << representative(it) << std::endl;
        if (infix(representative(it), parentRepLength(it) + 1, endPos) !=
            infix(pattern, parentRepLength(it) + 1, endPos))
            return 0;
    }

    // if we get here the pattern was found
    // output match positions
    for (unsigned i = 0; i < length(getOccurrences(it)); ++i)
        std::cout << getOccurrences(it)[i] << std::endl;

    return 0;
}
```

Afterwards we create the TopDown Iterator using the metafunction Iterator, which expects two arguments, the type of the container to be iterated and a specialization tag (see the VSTree Iterator hierarchy and the *Iteration* Tutorial for more details).

```
    Iterator<TIndex, TopDown<> >::Type it(index);
```

The main search can then be implemented using the functions repLength and representative. Since goDown might cover more than one character (when traversing trees) it is necessary to compare parts of the pattern against the representative of the iterator. The search can now be implemented as follows. The algorithm descends the suffix tree along edges beginning with the corresponding pattern character. In each step the `unseen` edge characters have to be verified.

```
    CharString pattern = "wood";
    while (repLength(it) < length(pattern))
```

```
    {
        // go down edge starting with the next pattern character
        if (!goDown(it, pattern[repLength(it)]))
            return 0;

        unsigned endPos = std::min((unsigned)repLength(it),␣
↪(unsigned)length(pattern));
        // compare remaining edge characters with pattern
        std::cout << representative(it) << std::endl;
        if (infix(representative(it), parentRepLength(it) + 1, endPos) !=
            infix(pattern, parentRepLength(it) + 1, endPos))
            return 0;
    }
```

If all pattern characters could successfully be compared we end in the topmost node who's leaves point to text positions starting with the pattern. Thus, the suffixes represented by this node are the occurrences of our pattern and can be retrieved with getOccurrences.

```
    // if we get here the pattern was found
    // output match positions
    for (unsigned i = 0; i < length(getOccurrences(it)); ++i)
        std::cout << getOccurrences(it)[i] << std::endl;

    return 0;
}
```

Program output:

```
w
wo
wood
9
22
```

Alternatively, we could have used goDown to go down the path of the entire pattern instead of a single characters:

```
    if (goDown(it, "wood"))
        for (unsigned i = 0; i < length(getOccurrences(it)); ++i)
            std::cout << getOccurrences(it)[i] << std::endl;

    return 0;
}
```

```
9
22
```

Tip: When implementing recursive algorithms such as an approximate search using backtracking, we recommend the use of the TopDownIterator without history. By passing the iterator by value, the history is stored implicitly on the call stack.

Assignment 1

Type Review

Objective Copy the code into a demo program and replace the text with a string set containing the strings `"How much"`, `"wood would"` and `" a woodchuck chuck?"`.

Solution

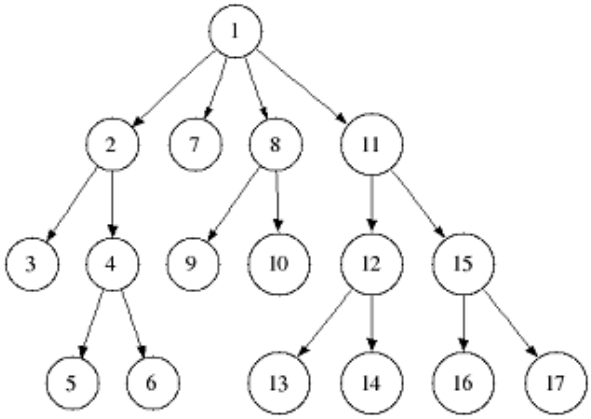
```
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    StringSet<String<char> > text;
    appendValue(text, "How much");
    appendValue(text, " wood would");
    appendValue(text, " a woodchuck chuck?");

    typedef Index<StringSet<String<char> > > TIndex;
    TIndex index(text);
    Iterator<TIndex, TopDown<> >::Type it(index);

    CharString pattern = "wood";
    while (repLength(it) < length(pattern))
    {
        // go down edge starting with the next pattern character
        if (!goDown(it, pattern[repLength(it)]))
            return 0;

        unsigned endPos = _min(repLength(it), length(pattern));
        // compare remaining edge characters with pattern
        std::cout << representative(it) << std::endl;
        if (infix(representative(it), parentRepLength(it) + 1, endPos) !=
            infix(pattern, parentRepLength(it) + 1, endPos))
            return 0;
    }

    // if we get here the pattern was found
    // output match positions
    for (unsigned i = 0; i < length(getOccurrences(it)); ++i)
        std::cout << getOccurrences(it)[i] << std::endl;

    return 0;
}
```

The difference is the format of the positions of the found occurrences. Here, we need a Pair to indicate the string within the StringSet and a position within the string.

Assignment 2

Type Review

Objective Write a program that traverses the nodes of the suffix tree of `"tobeornottobe"` in the order shown here:



At each node print the text of the edges from the root to the node. You may only use the functions goDown, goRight, goUp and isRoot to navigate and representative which returns the string that represents the node the iterator points to.

Hint

- Use a TopDown History Iterator.
- The code skeleton could look like this:

```
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main()
{
        typedef Index<CharString> TIndex;
        TIndex index("tobeornottobe");
        Iterator< TIndex, TopDown<ParentLinks<> > >::Type it(index);
/*
        do {
                //...
        } while (isRoot(it));
*/
        return 0;
}
```

Solution One iteration step of a preorder DFS can be described as follows:

- if possible, go down one node
- if not:
  - if possible, go to the next sibling
  - if not:
    - go up until it is possible to go to a next sibling
    - stop the whole iteration after reaching the root node

Thus, the DFS walk can be implemented in the following way:

```
#include <iostream>
#include <seqan/index.h>
```

```
using namespace seqan;

int main()
{
    typedef Index<CharString> TIndex;
    TIndex index("tobeornottobe");
    Iterator<TIndex, TopDown<ParentLinks<> > >::Type it(index);

    do
    {
        std::cout << representative(it) << std::endl;
        if (!goDown(it) && !goRight(it))
            while (goUp(it) && !goRight(it))
                ;
    }
    while (!isRoot(it));

    return 0;
}
```

Assignment 3

Type Review

Objective Modify the program to efficiently skip nodes with representatives longer than 3. Move the whole program into a template function whose argument specifies the index type and call this function twice, once for the IndexEsa and once for the IndexWotd index.

Solution We modify the DFS traversal to skip the descent if we walk into a node whose representative is longer than 3. We then proceed to the right and up as long as the representative is longer than 3.

```
template <typename TIndexSpec>
void constrainedDFS()
{
    typedef Index<CharString, TIndexSpec> TIndex;
    TIndex index("tobeornottobe");
    typename Iterator<TIndex, TopDown<ParentLinks<> > >::Type it(index);

    do
    {
        std::cout << representative(it) << std::endl;
        if (!goDown(it) || repLength(it) > 3)
            do
            {
                if (!goRight(it))
                    while (goUp(it) && !goRight(it))
                        ;
            }
            while (repLength(it) > 3);
    }
    while (!isRoot(it));
    std::cout << std::endl;
}

int main()
{
    constrainedDFS<IndexEsa<> >();
```

```
    constrainedDFS<IndexWotd<> >();
    return 0;
}
```

```
be
e
o
obe
t


be
e
o
obe
t
```

Depth-First Search

The tree traversal in assignment 2 is equal to a the tree traversal in a full depth-first search (dfs) over all suffix tree nodes beginning either in the root (preorder dfs) or in a leaf node (postorder dfs). A preorder traversal (*Preorder DFS*) halts in a node when visiting it for the first time whereas a postorder traversal (*Postorder DFS*) halts when visiting a node for the last time. The following two figures give an example in which order the tree nodes are visited.

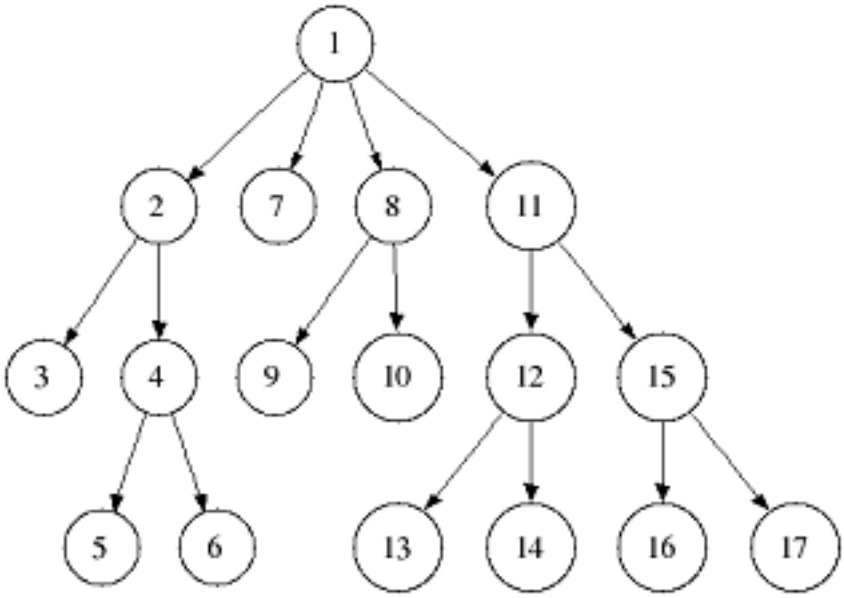


Fig. 4.1: Preorder DFS

Since these traversals are frequently needed SeqAn provides special iterators which we will describe next.

We want to construct the suffix tree of the string “abracadabra” and output the substrings represented by tree nodes in preorder dfs. In order to do so, we create the string “abracadabra” and an index specialized with the type of this string.

The Iterator metafunction expects two arguments, the type of the container to be iterated and a specialization tag, as described earlier. In this example we chose a TopDown History Iterator whose signature in the second template argument is `TopDown< ParentLinks<Preorder> >`.

```
↪myString = "abracadabra
↪";


↪<char> > TMyIndex;

↪myIndex(myString);


↪TopDown<ParentLinks
↪<Preorder> > >::Type␣
↪myIterator(myIndex);


↪atEnd(myIterator))


↪representative(myIterator)␣
↪<< std::endl;
```

As all DFS suffix tree iterators implement the VSTree Iterator, they can be used via goNext, atEnd, etc.

```
    Iterator<TMyIndex, TopDown<ParentLinks<Preorder> > >::Type myIterator(myIndex);

    while (!atEnd(myIterator))
    {
        std::cout << representative(myIterator) << std::endl;
        ++myIterator;
    }

    return 0;
}
```

Program output:

```
a
abra
abracadabra
acadabra
adabra
bra
bracadabra
cadabra
dabra
ra
racadabra
```

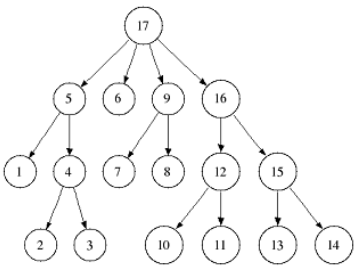


Fig. 4.2: Postorder DFS

Tip: There are currently 2 iterators in SeqAn supporting a DFS search:

Iterator	Preorder	Postorder
BottomUpIterator	no	yes
TopDownHistoryIterator	yes	yes

If solely a postorder traversal is needed the BottomUp Iterator should be preferred as it is more memory efficient. Please note that the BottomUp Iterator is only applicable to IndexEsa indices.

Tip: A relaxed suffix tree (see *Indices*) is a suffix tree after removing the $ characters and empty edges. For some bottom-up algorithms it would be better not to remove empty edges and to have a one-to-one relationship between leaves and suffices. In that cases you can use the tags PreorderEmptyEdges or PostorderEmptyEdges instead of Preorder or Postorder or EmptyEdges for the TopDown Iterator.

Note that the goNext is very handy as it simplifies the tree traversal in assignment 2 greatly.

Assignment 4

Type Review

Objective Write a program that constructs an index of the StringSet "tobeornottobe", "thebeeonthecomb", "beingjohnmalkovich" and outputs the strings corresponding to suffix tree nodes in postorder DFS.

Solution First we have to create a StringSet of CharString (shortcut for `String<char>`) and append the 3 strings to it. This could also be done by using resize and then assigning the members with `operator[]`. The first template argument of the index class has to be adapted and is now a StringSet.

```
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    StringSet<CharString> myStringSet;
    appendValue(myStringSet, "tobeornottobe");
    appendValue(myStringSet, "thebeeonthecomb");
    appendValue(myStringSet, "beingjohnmalkovich");

    typedef Index<StringSet<CharString> > TMyIndex;
    TMyIndex myIndex(myStringSet);
```

To switch to postorder DFS we have to change the specialization tag of `ParentLinks` from `Preorder` to `Postorder`. Please note that the TopDownHistoryIterator always starts in the root node, which is the last postorder DFS node. Therefore, the iterator has to be set explicitly to the first DFS node via goBegin.

```
    Iterator<TMyIndex, TopDown<ParentLinks<Postorder> > >::Type␣
↪myIterator(myIndex);

    // Top-down iterators start in the root node which is not the first node of a
    // postorder DFS. Thus we have to manually go to the DFS start with goBegin
    goBegin(myIterator);
    while (!atEnd(myIterator))
    {
        std::cout << representative(myIterator) << std::endl;
```

```
        ++myIterator;
    }
```

Alternatively with a TopDownHistoryIterator you also could have used a BottomUpIterator with the same result. The BottomUp Iterator automatically starts in the first DFS node as it supports no random access.

```
    Iterator<TMyIndex, BottomUp<> >::Type myIterator2(myIndex);

    while (!atEnd(myIterator2))
    {
        std::cout << representative(myIterator2) << std::endl;
        ++myIterator2;
    }

    return 0;
}
```

Program output:

```
alkovich
beeonthecomb
beingjohnmalkovich
beornottobe
be
b
ch
comb
c
ebeeonthecomb
ecomb
eeonthecomb
eingjohnmalkovich
eonthecomb
eornottobe
eo
e
gjohnmalkovich
hebeeonthecomb
hecomb
he
hnmalkovich
h
ich
ingjohnmalkovich
i
johnmalkovich
kovich
lkovich
malkovich
mb
m
ngjohnmalkovich
nmalkovich
nottobe
nthecomb
n
obeornottobe
obe
```

```
ohnmalkovich
omb
onthecomb
ornottobe
ottobe
ovich
o
rnottobe
thebeeonthecomb
thecomb
the
tobeornottobe
tobe
ttobe
t
vich

alkovich
beeonthecomb
beingjohnmalkovich
beornottobe
be
b
ch
comb
c
ebeeonthecomb
ecomb
eeonthecomb
eingjohnmalkovich
eonthecomb
eornottobe
eo
e
gjohnmalkovich
hebeeonthecomb
hecomb
he
hnmalkovich
h
ich
ingjohnmalkovich
i
johnmalkovich
kovich
lkovich
malkovich
mb
m
ngjohnmalkovich
nmalkovich
nottobe
nthecomb
n
obeornottobe
obe
ohnmalkovich
omb
```

```
onthecomb
ornottobe
ottobe
ovich
o
rnottobe
thebeeonthecomb
thecomb
the
tobeornottobe
tobe
ttobe
t
vich
```

As the last assignment lets try out one of the specialized iterators, which you can find at the bottom of this page. Look there for the specialization which iterates over all maximal unique matches (MUMS).

Assignment 5

Type Review

Objective Write a program that outputs all maximal unique matches (MUMs) between `"CDFGHC"` and `"CDEFGAHC"`.

Solution Again, we start to create a StringSet of CharString and append the 2 strings.

```
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    StringSet<CharString> myStringSet;
    appendValue(myStringSet, "CDFGHC");
    appendValue(myStringSet, "CDEFGAHC");

    typedef Index<StringSet<CharString> > TMyIndex;
    TMyIndex myIndex(myStringSet);
```

After that we simply use the predefined iterator for searching MUMs, the MumsIterator. Its constructor expects the index and optionally a minimum MUM length as a second parameter. The set of all MUMs can be represented by a subset of suffix tree nodes. The iterator will halt in every node that is a MUM of the minimum length. The corresponding match is the node's representative.

```
    Iterator<TMyIndex, Mums>::Type myIterator(myIndex);

    while (!atEnd(myIterator))
    {
        std::cout << representative(myIterator) << std::endl;
        ++myIterator;
    }

    return 0;
}
```

Program output:

```
CD
FG
HC
```

Accessing Suffix Tree Nodes

In the previous subsection we have seen how to walk through a suffix tree. We now want to know what can be done with a suffix tree iterator. As all iterators are specializations of the general VSTree Iterator class, they inherit all of its functions. There are various functions to access the node the iterator points at (some we have already seen), so we concentrate on the most important ones.

representative returns the substring that represents the current node, i.e. the concatenation of substrings on the path from the root to the current node

getOccurrence returns a position where the representative occurs in the text

getOccurrences returns a string of all positions where the representative occurs in the text

isRightTerminal tests if the representative is a suffix in the text (corresponds to the shaded nodes in the *Indices* figures)

isLeaf tests if the current node is a tree leaf

parentEdgeLabel returns the substring that represents the edge from the current node to its parent (only TopDownHistory Iterator)

Important: There is a difference between the functions isLeaf and isRightTerminal. In a relaxed suffix tree (see *Indices*) a leaf is always a suffix, but not vice versa, as there can be internal nodes a suffix ends in. For them isLeaf returns false and isRightTerminal returns true.

Property Maps

Some algorithms require to store auxiliary information (e.g. weights, scores) to the nodes of a suffix tree. To attain this goal SeqAn provides so-called property maps, simple Strings of a property type. Before storing a property value, these strings must first be resized with resizeVertexMap. The property value can then be assigned or retrieved via assignProperty, getProperty, or property. It is recommended to call resizeVertexMap prior to every call of assignProperty to ensure that the property map has sufficient size. The following example iterates over all nodes in preorder dfs and recursively assigns the node depth to each node. First we create a String of `int` to store the node depth for each suffix tree node.

```
int main()
{
    String<char> myString = "abracadabra";

    typedef Index<String<char>, IndexWotd<> > TMyIndex;
    TMyIndex myIndex(myString);
    String<int> propMap;
```

The main loop iterates over all nodes in preorder DFS, i.e. parents are visited prior children. The node depth for the root node is 0 and for all other nodes it is the parent node depth increased by 1. The functions assignProperty, getProperty and property must be called with a VertexDescriptor. The vertex descriptor of the iterator node is returned by value and the descriptor of the parent node is returned by nodeUp.

```
    Iterator<TMyIndex, TopDown<ParentLinks<Preorder> > >::Type myIterator(myIndex);

    int depth;
    while (!atEnd(myIterator))
    {
        if (isRoot(myIterator))
            depth = 0;
        else
            depth = getProperty(propMap, nodeUp(myIterator)) + 1;

        resizeVertexMap(propMap, myIndex);
        assignProperty(propMap, value(myIterator), depth);

        ++myIterator;
    }
```

At the end we again iterate over all nodes and output the calculated node depth.

```
    goBegin(myIterator);
    while (!atEnd(myIterator))
    {
        std::cout << getProperty(propMap, value(myIterator)) << '\t' <<␣
↪representative(myIterator) << std::endl;
        ++myIterator;
    }
    return 0;
}
```

Program output:

```
0
1        a
2        abra
3        abracadabra
2        acadabra
2        adabra
1        bra
2        bracadabra
1        cadabra
1        dabra
1        ra
2        racadabra
```

Tip: In SeqAn there is already a function nodeDepth defined to return the node depth.

Additional iterators

By now, we know the following iterators (n = text size, σ = alphabet size, d = tree depth):

Iterator specialization	Description	Space	Index tables
BottomUpIterator	postorder dfs	$\mathcal{O}(d)$	SA, LCP
TopDownIterator	can go down and go right	$\mathcal{O}(1)$	SA, Lcp, Childtab
TopDownHistoryIterator	can also go up, preorder/postorder dfs	$\mathcal{O}(d)$	SA, Lcp, Childtab

Besides the iterators described above, there are some application-specific iterators in SeqAn:

Iterator specialization	Description	Space	Index tables
MaxRepeatsIterator	maximal repeats	$\mathcal{O}(n)$	SA, Lcp, Bwt
SuperMaxRepeatsIterator	supermaximal repeats	$\mathcal{O}(d+\sigma)$	SA, Lcp, Childtab, Bwt
SuperMaxRepeatsFastIterator	supermaximal repeats (optimized for ESA)	$\mathcal{O}(\sigma)$	SA, Lcp, Bwt
MumsIterator	maximal unique matches	$\mathcal{O}(d)$	SA, Lcp, Bwt
MultiMemsIterator	multiple maximal exact matches (w.i.p.)	$\mathcal{O}(n)$	SA, Lcp, Bwt

Given a string s a repeat is a substring r that occurs at 2 different positions i and j in s. The repeat can also be identified by the triple (i,j,|r|). A maximal repeat is a repeat that cannot be extended to the left or to the right, i.e. s[i-1]s[j-1] and s[i+|r|]s[j+|r|]. A supermaximal repeat r is a maximal repeat that is not part of another repeat. Given a set of strings $s_1, \ldots, s_m$ a MultiMEM (multiple maximal exact match) is a substring r that occurs in each sequence s_i at least once and cannot be extended to the left or to the right. A MUM (maximal unique match) is a MultiMEM that occurs exactly once in each sequence. The following examples demonstrate the usage of these iterators:

- Demo Maximal Unique Matches
- Demo Supermaximal Repeats
- Demo Maximal Repeats

Indices in SeqAn allow efficient pattern queries in strings or sets of strings. In contrast to, e.g., online-search algorithms that search through the text in $\mathcal{O}(n)$, substring indices find a pattern in sublinear time $o(n)$.

Indices store additional data structures that allow searching the text using an iterator. Using the iterator can be thought of as traversing a suffix tree. The following section gives you an introduction how the suffix tree is built.

Tip: The Finder interface allows searching indices without using the iterator. For more information check out the tutorial on *Indexed Pattern Matching*.

Suffix Trees

We consider an alphabet Σ and a sentinel character \$ that is smaller than every character of Σ. A suffix tree of a given non-empty string s over Σ is a directed tree whose edges are labeled with non-empty substrings of s\$ with the following properties:

1. Each outgoing edge begins with a different letter and the outdegree of an internal node is greater than 1. 2. Each suffix of s\$ is the concatenation of edges from the root to a leaf node. 3. Each path from the root to a leaf node is a suffix of s\$.

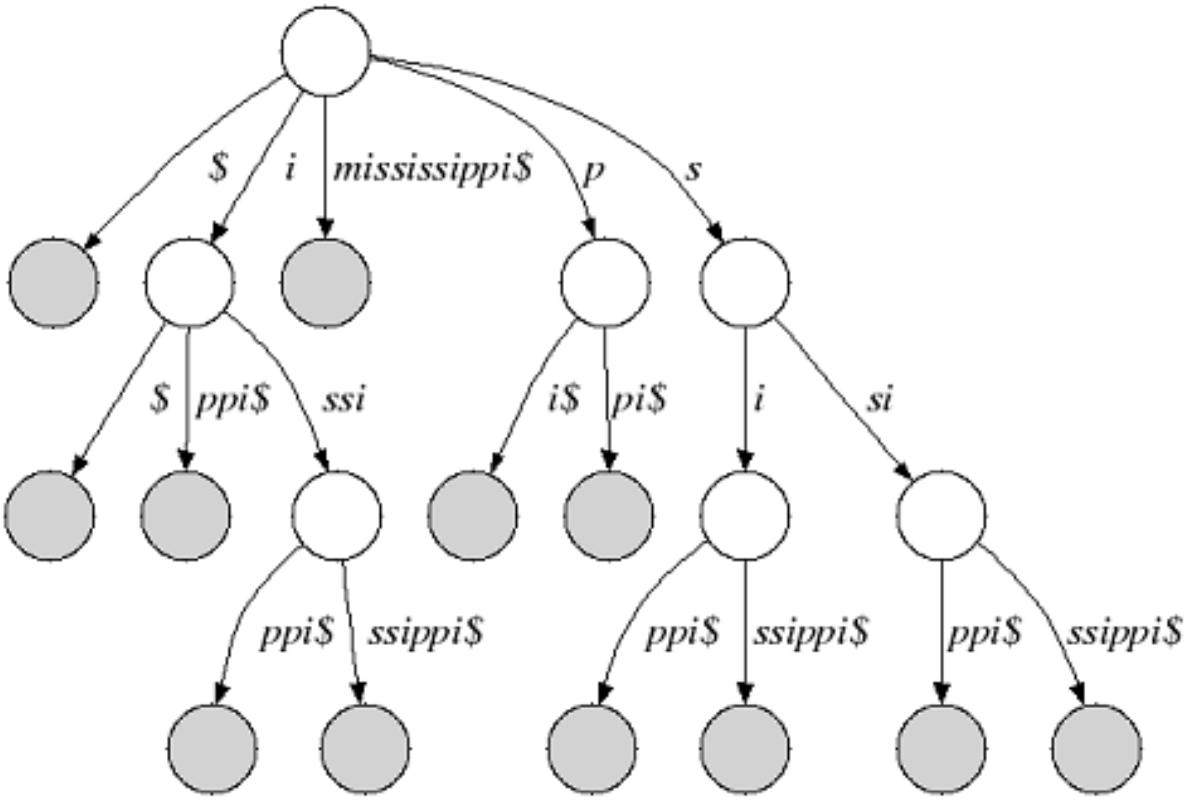
The following figure shows the suffix tree of the string s="mississippi" (suffix nodes are shaded):

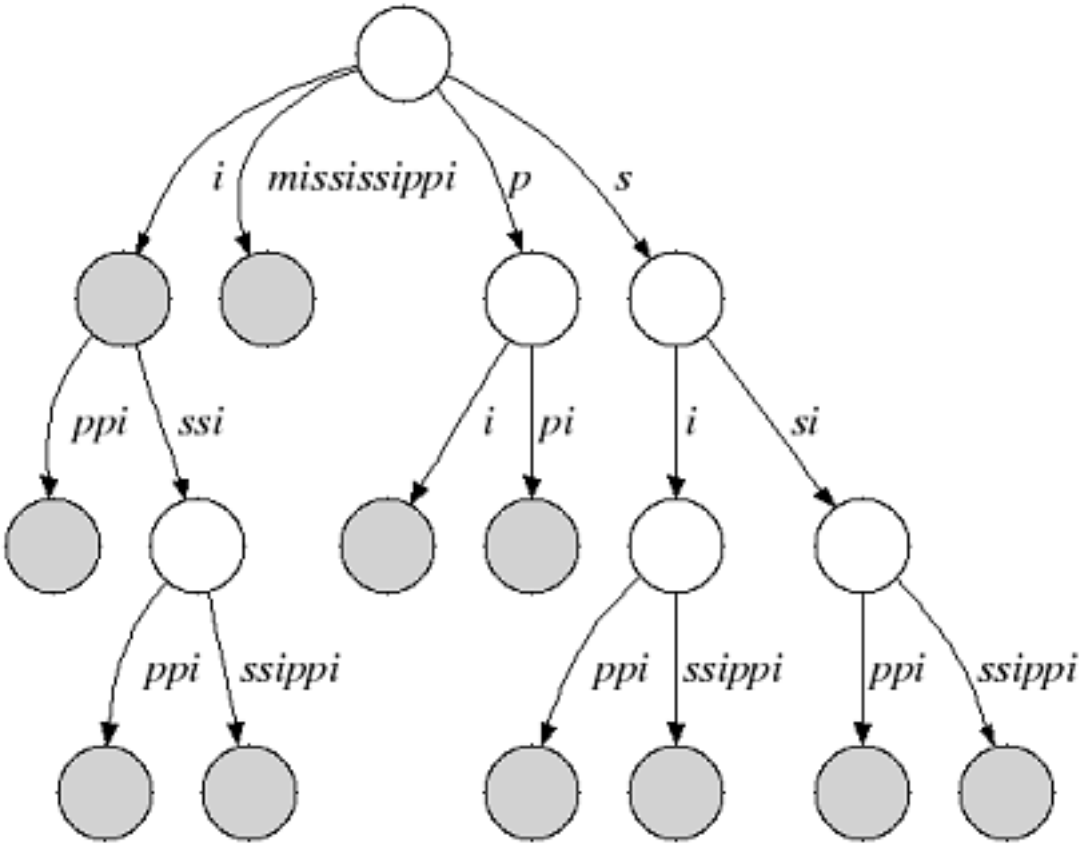
Figure 1: Suffix tree of "mississippi"

Many suffix tree construction algorithms expect \$ to be part of the string alphabet which is undesirable for small bit-compressible alphabets (e.g. DNA). In SeqAn there is no need to introduce a \$. We relax suffix tree criterion 2. and consider the relaxed suffix tree that arises from the suffix tree of s by removing the \$ character and all empty edges. In the following, we only consider relaxed suffix trees and simply call them suffix trees. In that tree a suffix can end in an inner node as you can see in the next figure (suffix "i"):

Figure 2: Relaxed suffix tree of "mississippi"

Alignment


$
i
mississippi$
p
s
$
ppi$
ssi
i$
pi$
i
si
ppi$
ssippi$
ppi$
ssippi$
ppi$
ssippi$


i
mississippi
p
s
ppi
ssi
i
pi
i
si
ppi
ssippi
ppi
ssippi
ppi
ssippi

ToC

Contents

Scoring Schemes

Learning Objective This tutorial introduces you to the scoring systems that can be used in SeqAn to quantify the sequence similarity. You will learn basic techniques to create and modify standard and custom scoring systems capable to satisfy the requirements of a wide range of applications.

Difficulty Basic

Duration 45 min

Prerequisites *A First Example*, *Sequences*, *Alignment Representation (Gaps)*

The alignment procedures are usually based on the sequences similarity computation described by an alignment scoring system that gives countable information used to determine which sequences are related and which are not.

Four main biological events must be considered during the sequence alignment: Conservation, substitution, insertion and deletion. We could have a Conservation when the two compared letters are the same and a Match is detected, a Substitution when we detect a Mismatch where a letter is aligned with another, and Insertion or Deletion when in one of the two aligned sequences a letter is aligned with a Gap. Matches, mismatches and gaps detected during the alignment do not guarantee to be the most representative biological truth since their dispositions is dependent of the chosen scoring schemes and the selected alignment algorithm. In order to improve the correlation between computed sequence alignment and biological similarity, specific combinations of scoring schemes and alignment algorithms have been developed during the years and are usually adopted for the alignment of different types of biological sequences. For example, as we will see in the following, the small RNA sequences are usually aligned with a Global Alignment algorithm implementing a Simple Score scheme, differently from the protein sequences that are mostly aligned with the Local Alignment algorithm that uses a Substitution Matrix Score scheme.

(Qry)	A C D E F G	A C D E F G	A C D E F G	A C -- E F G
(Sbj)	A C D E F G	A C L E F G	A C -- E F G	A C D E F G
Biological event	**Conservation**	**Substitution**	**Insertion**	**Deletion**
Alignment represent	**Match**	**Mismatch**	**Gap**	**Gap**

Given an alignment structure that store the two sequences and a scoring scheme, the score of the alignment can be computed as the sum of the scores for aligned character pairs plus the sum of the scores for all gaps.

With refer to the alignment procedure a Scoring Scheme can be defined as the set of rules used to assess the possible biological events that must be considered during the alignment procedure.

In SeqAn are available several scoring schemes to evaluate matches and mismatches, while three different gap models can be applied to consider insertions and deletions events. We will first introduce you to the scoring schemes used to evaluate match and mismatch. Subsequently, you will learn how to chose the gap model to be implemented in the chosen scoring scheme.

Match/Mismatch Evaluation

Simple Score

The simplest example of Scoring Scheme, usually applied to score the similarity among nucleotide sequences, is the Levenshtein distance model that assigns a score of 0 and -1 respectively if a match or a mismatch occurs, whereas a penalty value equal to -1 in case of gaps representing insertions or deletions (this scoring scheme is the default for SimpleScore). Alternatively, also the Hamming distance model can be used for some simple tasks that do not require the gap evaluations.

Now, let's start by constructing our first scoring function for the global alignment algorithm called with the function globalAlignment. As first step we need to include the header file `<seqan/align.h>` which contains the necessary data structures and functions associated with the alignments. The next steps would be to implement the main function of our program and to define the types that we want to use.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<AminoAcid> TSequence;                // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;         // align type
```

We first define the type of the input sequences (`TSequence`) and an Align object (`TAlign`) type to store the alignment. For more information on the Align datastructure, please read the tutorial *Alignment Representation (Gaps)*. After defining the types, we can continue to construct our own Align object. First, we create two input sequences `seq1 = "TELKDD"` and `seq2 = "LKTEL"`, then we define the scoring values for match, mismatch, gap. As last we create the 'align' object and resize it to manage two Gaps objects, at this point we filled it with the sequences to be aligned.

```
    TSequence seq1 = "TELKDD";
    TSequence seq2 = "LKTEL";
    int match = -0;
    int mismatch = -1;
    int gap = -1;

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);
```

Now, we can compute the global alignment that makes use of the simple scoring function. To do so, we simply call the function globalAlignment and give as input parameters the `align` object and the scoring scheme representing the Levenshtein distance. The globalAlignment function fills the `align` object with the best computed alignment and returns the maximum score which we store in the `score` variable. Afterwards, we print the computed score and the corresponding alignment.

```
    int score = globalAlignment(align, Score<int, Simple>(match, mismatch, gap));
    std::cout << "Score: " << score << std::endl;
    std::cout << align << std::endl;

    return 0;
}
```

Congratulations! You have created your global alignment implementing the simple scoring function, the output is as follows:

```
Score: -5
      0     .
        TELK-DD
          ||
        --LKTEL
```

However, in the evaluation of protein similarity or for advanced nucleotide alignments a more complex scoring model is generally applied. It is based on the usage of a Substitution Matrix, proven to better describe from a biological point of view, events such as matches and mismatches.

Substitutional Matrices Score

Substitutional Matrices are built on the basis of the probability that a particular amino acid or nucleotide is replaced with another during the evolution process. They assign to each pair a value that indicates their degree of similarities, obtained thanks to statistical methods reflecting the frequency of a particular substitution in homologous protein or RNA families. A positive value in the Substitutional Matrix means that the two letters share identical or similar properties.

These scoring schemes store a score value for each pair of characters. This value can be accessed using score. Examples for this kind of scoring scheme are Pam120 and Blosum62. Anyway the class MatrixScore can be used to store arbitrary scoring matrices for the creation of custom scoring systems, as shown in the example proposed in the *Working With Custom Score Matrices*.

Blosum matrix, is one of the most used Substitutional Matrix implemented by considering multiple alignments of evolutionarily divergent proteins, while Ribosum is the RNA counterpart computed using ribosomal sequences.

In the following example it is proposed the construction of a scoring function for a global alignment algorithm that uses the Blosum62 matrix to score the matched and mismatched letters. As first we include the header file `<seqan/align.h>` which contains the necessary data structures and functions associated with the alignments, then we implement the main function of our program and define the types that we want to use.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<AminoAcid> TSequence;                // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;         // align type
```

The input sequences type `TSequence` and the Align object of type `TAlign` are defined and the two input sequences `seq1 = "TELKDD"` and `seq2 = "LKTEL"` together with the gap penalty are assigned. In this case we define only the gap value since the Blosum matrix will be used to score matches and mismatches. Then the sequences are associated with the alignment object.

```
    TSequence seq1 = "TELKDD";
    TSequence seq2 = "LKTEL";
    int gap = -1;

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);
```

Now, we compute the global alignment function, providing as second parameter the tag referred to the Blosum62 matrix together with the gap costs. To do so, we simply call the function globalAlignment and give as input parameters the `align` object and the Blosum62 scoring scheme. The globalAlignment function returns the score of the best alignment, which we store in the `score` variable that is then printed together with the corresponding alignment.

```
    int score = globalAlignment(align, seqan::Blosum62(gap, gap));
    std::cout << "Score: " << score << std::endl;
    std::cout << align << std::endl;

    return 0;
}
```

The output of a global alignment implementing the Blosum62 scoring function is as follows:

```
Score: 9
      0     .
        --TELKDD
          |||
        LKTEL---
```

Note: As can be noted the output of this scoring scheme is completely different with respect to the output generated with the simple scoring scheme confirming that the scoring scheme choice is one of the most important step to achieve high quality alignments.

Gap Evaluation

In the previous sections we proposed two simple code examples useful to highlight the differences between two scoring schemes capable to evaluate match and mismatch events. In this section we will see the three gap models, implemented in the SeqAn library, to evaluate the insertion and deletion events.

Linear Gap Model

The easiest is the Linear gap model that considers, for the alignment score computation, the gap length (g) giving the possibility to evaluate with different scores gaps of different sizes;

$$\gamma_{lin}(g) = -g * d$$

This gap model is chosen as standard when only a gap value is provided in the scoring function or when the two provided gaps have the same value. For instance, this gap model as been adopted during the alignment computation of the two proposed examples.

Affine Gap Model

It has been proven that the first amino acid or nucleotide inserted/deleted (identified as gap open) found during the alignment operations is more significant, from a biological point of view, than the subsequent ones (called gap extension), making the so called Affine Gap model a viable solution for the alignment of biomolecules *[Car06]*. Affine gap model that attribute different costs to the gap open (d) and the gap extension (e) events, is able to assign an higher penalty to the gap presence with respect to its relative length (g).

$$\gamma_{aff}(g) = -d - (g-1) * e$$

The Affine Gap model implemented in the DP alignment algorithms is however quite expensive both in terms of computational time as well as in terms of memory requirements with respect to other less demanding solutions such as the Linear Gap model application.

Dynamic Gap Model

In SeqAn is provided an optimised version of the Affine Gap model called Dynamic Gap Selector (DGS) designed by Urgese et al. *[UPA+14]*. This new gap model can be used to reduce the computational time and the memory requirement while keeping the alignment scores close to those computed with the Affine Gap model. The usage of Dynamic Gap model in the Global alignment computation of long strings can give results slightly different from those computed using Affine Gap model since the alignment matrix became bigger and different alignment paths can be chosen during the alignment procedure. Score variation are rare when Dynamic Gap model is used in the Local alignments.

Example Affine vs Dynamic

Tip: The order of the different costs in the scoring scheme is `match`, `mismatch`, `gapExtend` and `gapOpen`. The gap model selection can be done providing one of the three specific tags (`LinearGaps()`, `AffineGaps()` or `DynamicGaps()`) as last parameter in the scoring function creation. If you want to use Linear Gap costs you could also omit the last parameter `gapOpen` and the scoring scheme would automatically choose the Linear Gap cost function. The Affine Gap model is chosen as standard when the gap costs are different and the gap model tag is not provided. If the Dynamic Gap model is required the relative tag must be supplied.

In the following we propose an example where two different scoring functions have been created to show how to call a global alignment algorithm that uses the Blosum62 plus the `AffineGaps()` and `DynamicGaps()` specializations. The inclusion of the header and the type definition is identical to the previous examples.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<AminoAcid> TSequence;                // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;         // align type
```

The input sequences type and the Align object of type `TAlign` are then create and initialized. As can be noted we define two different gap values, one for the gap extension and one for the gap open. Even in this example the Blosum62 will be used to score match and substitutions events.

```
    TSequence seq1 = "TELKDD";
    TSequence seq2 = "LKTEL";
    int gapExtend = -2;
    int gapOpen = -10;


    TAlign alignAffine;
    resize(rows(alignAffine), 2);
    assignSource(row(alignAffine, 0), seq1);
    assignSource(row(alignAffine, 1), seq2);

    TAlign alignDynamic;
    resize(rows(alignDynamic), 2);
    assignSource(row(alignDynamic, 0), seq1);
    assignSource(row(alignDynamic, 1), seq2);
```

Now, we can compute the global alignment function providing as second parameter the tag referred to the Blosum62 matrix filled with the two different gap costs. Moreover, the tag for the gap model selection is provided. To do so, we simply call the function globalAlignment and give as input parameters the `align` object, the Blosum62 scoring scheme and the `AffineGaps()` or `DynamicGaps()` tag. The globalAlignment function output is then printed.

```
    int scoreAffine = globalAlignment(alignAffine, Blosum62(gapExtend, gapOpen),␣
↪AffineGaps());
    std::cout << "ScoreAffine: " << scoreAffine << std::endl;
    std::cout << alignAffine << std::endl;

    int scoreDynamic = globalAlignment(alignDynamic, Blosum62(gapExtend, gapOpen),␣
↪DynamicGaps());
    std::cout << "ScoreDynamic: " << scoreDynamic << std::endl;
    std::cout << alignDynamic << std::endl;

    return 0;
}
```

The output of a global alignment implementing the Blosum62 with the two gap models is as follows:

```
ScoreAffine: -12
      0     .
        TELKDD-
          ||
        --LKTEL



ScoreDynamic: -12
      0     .
        TELKDD-
          ||
        --LKTEL
```

Tip: The functions scoreMatch and scoreMismatch access values for match and mismatch. The function scoreGap, or scoreGapExtend and scoreGapOpen access values for gaps.

ToC

Contents

Alignment Representation (Gaps)

Learning Objective This tutorial introduces you to the gaps data structures that can be used to represent an alignment in SeqAn. You will learn basic techniques to create and modify such data structures and how to access certain information from these data structures.

Difficulty Basic

Duration 15 min

Prerequisites *A First Example*, *Sequences*

The Align data structure is simply a set of multiple Gaps data structures. A Gaps data structure is a container storing gap information for a given source sequence. The gap information is put on top of the source sequence (coordinates of the gapped sequence refer to the **gap space**) without directly applying them to the source (coordinates of the ungapped sequence refer to the **source space**). This way operating with gaps sustains very flexible.

Gaps data structures

There are two specializations available for the Gaps data structures: Array Gaps and Anchor Gaps. They differ in the way they implement the gap space.

Note: In general, using Array Gaps is sufficient for most applications. This specialization is also the default one if nothing else is specified. It simply uses an array which stores the counts of gaps and characters in an alternating order. Thus, it is quite efficient to extend existing gaps while it is more expensive to search within the gapped sequence or insert new gaps. Alternatively, one should prefer Anchor Gaps if many conversions between coordinates of the gap and the source space are needed as binary search can be conducted to search for specific positions.

Constructing an alignment

Now, let's start by constructing our first alignment. Before we can make use of any of the mentioned data structures, we need to tell the program where to find the definitions. This can be achieved by including the header file `<seqan/align.h>` which contains the necessary data structures and functions associated with the alignments. The next steps would be to implement the main function of our program and to define the types that we want to use.

```
#include <iostream>
#include <seqan/align.h>
```

```
using namespace seqan;

int main()
{
```

We first define the type of the input sequences (`TSequence`). Then we can define the type of our actual Align object we want to use. In an Align object, the gapped sequences are arranged in rows. You can use the Metafunction Row to get the correct type of the used Gaps objects. In the following we use the term `row` to explicitly refer to a single gapped sequence in the Align object. We will use the term `gapped sequence` to describe functionalities that are related to the Gaps data structure independent of the Align object.

```
    typedef char TChar;                                 // character type
    typedef String<TChar> TSequence;                    // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;         // align type
    typedef Row<TAlign>::Type TRow;                     // gapped sequence type
```

After defining the types, we can continue to actually construct our own Align object. Therefore, we need to resize the alignment object in order to reserve space for the sequences we want to add. In our case, we assume a pairwise alignment, hence we reserve space for 2 sequences. With the function row, we get access to the gapped sequence at a specific row in the alignment object. This is similar to the value function used in String Sets. Now, we can assign the source to the corresponding gapped sequence.

```
    TSequence seq1 = "CDFGDC";
    TSequence seq2 = "CDEFGAHGC";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);
    std::cout << align;
```

```
    0     .
      CDFGDC
      ||
      CDEFGA
```

Note: The second string `CDEFGAHGC` of the alignment is cropped in the output to `CDEFGA`, such that they are of equal length. Note that the string itself is not modified, i.e. not shortened.

After assigning the sources to the gapped sequences, we need to add some gaps to make it look like a real alignment. You can use the functions insertGap() and removeGap() to insert and delete one gap or insertGaps() and removeGaps() to insert and delete multiple gaps in a gapped sequence.

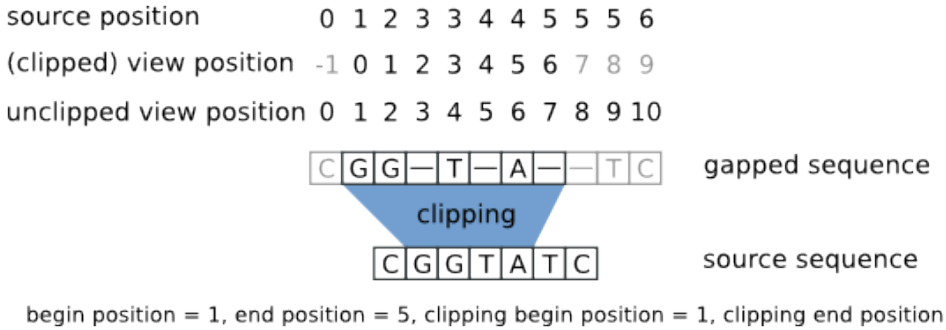
```
    TRow & row1 = row(align, 0);
    TRow & row2 = row(align, 1);
    insertGap(row1, 2);
    std::cout << align;
    insertGaps(row1, 5, 2);
    std::cout << align;
```

```
    0     .
      CD-FGDC
      || ||
      CDEFGAH
```

```
0     .
  CD-FG--DC
  || ||   |
  CDEFGAHGC
```

Congratulations! You have created your first alignment. Note that we used a reference declaration `TRow &` for the variables `row1` and `row2`. Without the reference, we would only modify copies of rows and the changes would not effect our `align` object.

Gap Space vs. Source Space



In the next steps, we want to dig a little deeper to get a feeling for the gap space and the source space. As mentioned above, the gaps are not inserted into the source but put on top of it in a separate space, the gap space. When inserting gaps, the gap space is modified and all coordinates right of the inserted gap are shifted to the right by the size of the gap. At the same time, the coordinates of the source remain unchanged. Using the function toSourcePosition(), we can determine which position of the current gapped sequence (gap space) corresponds to the position in the source space.

```
    std::cout << std::endl << "ViewToSource1: " << std::endl;
    for (auto c: row1)
        std::cout << c << " ";
    std::cout << std::endl;

    for (unsigned i = 0; i < length(row1); ++i)
        std::cout << toSourcePosition(row1, i) << " ";
    std::cout << std::endl;


    std::cout << std::endl << "ViewToSource2: " << std::endl;
    for (auto c: row2)
        std::cout << c << " ";
    std::cout << std::endl;

    for (unsigned i = 0; i < length(row2); ++i)
        std::cout << toSourcePosition(row2, i) << " ";
    std::cout << std::endl;
```

```
ViewToSource1:
C D - F G - - D C
0 1 2 2 3 4 4 4 5

ViewToSource2:
```

```
C D E F G A H G C
0 1 2 3 4 5 6 7 8
```

If the position in the gap space is actually a gap, then toSourcePosition() returns the source position of the next character to the right that is not a gap. Vice versa, we can determine where our current source position maps into the gap space using the function toViewPosition().

```
    std::cout << std::endl << "SourceToView1: " << std::endl;
    for (auto c: source(row1))
        std::cout << c << " ";
    std::cout << std::endl;

    for (unsigned i = 0; i < length(source(row1)); ++i)
        std::cout << toViewPosition(row1, i) << " ";
    std::cout << std::endl;


    std::cout << std::endl << "SourceToView2: " << std::endl;
    for (auto c: source(row2))
        std::cout << c << " ";
    std::cout << std::endl;

    for (unsigned i = 0; i < length(source(row2)); ++i)
        std::cout << toViewPosition(row2, i) << " ";
    std::cout << std::endl;
```

```
SourceToView1:
C D F G D C
0 1 3 4 7 8

SourceToView2:
C D E F G A H G C
0 1 2 3 4 5 6 7 8
```

In the first alignment, it seems that the end of the second row is cropped off to match the size of the first one. This effect takes place only in the visualization but is not explicitly applied to the gapped sequence. The second alignment is the one we manually constructed. Here, you can see that the second row is expanded to its full size while it matches the size of the first row. However, it is possible to explicitly crop off the ends of a gapped sequence by using the functions setClippedBeginPosition() and setClippedEndPosition(). These functions shrink the gap space and can be understood as defining an infix of the gapped sequence. After the clipping, the relative view position changes according to the clipping and so does the mapping of the source positions to the gap space. The mapping of the view positions to the source space does not change.

```
    std::cout << std::endl << "Before clipping:\n" << align;
    setClippedBeginPosition(row1, 1);
    setClippedEndPosition(row1, 7);
    setClippedBeginPosition(row2, 1);
    setClippedEndPosition(row2, 7);
    std::cout << std::endl << "After clipping:\n" << align;
```

```
Before clipping:
      0     .
        CD-FG--DC
        || ||   |
        CDEFGAHGC
```

```
After clipping:
      0     .
        D-FG--
        | ||
        DEFGAH
```

Here the output of the clipping procedure.

```
    std::cout << std::endl << "ViewToSource1: ";
    for (unsigned i = 0; i < length(row1); ++i)
        std::cout << toSourcePosition(row1, i) << " ";

    std::cout << std::endl << "ViewToSource2: ";
    for (unsigned i = 0; i < length(row2); ++i)
        std::cout << toSourcePosition(row2, i) << " ";
    std::cout << std::endl;

    std::cout << std::endl << "SourceToView1: ";
    for (unsigned i = 0; i < length(source(row1)); ++i)
        std::cout << toViewPosition(row1, i) << " ";

    std::cout << std::endl << "SourceToView2: ";
    for (unsigned i = 0; i < length(source(row2)); ++i)
        std::cout << toViewPosition(row2, i) << " ";
    std::cout << std::endl;
```

```
ViewToSource1: 1 2 2 3 4 4
ViewToSource2: 1 2 3 4 5 6

SourceToView1: -1 0 2 3 6 7
SourceToView2: -1 0 1 2 3 4 5 6 7
```

Note: It is important to understand the nature of the clipping information. It virtually shrinks the gap space not physically. That means the information before/after the begin/end of the clipping still exists and the physical gap space remains unchanged. To the outer world it seems the alignment is cropped off irreparably. But you can expand the alignment again by resetting the clipping information.

Iterating over Gapped Sequences

In the last part of this section, we are going to iterate over a Gaps object. This can be quite useful if one needs to parse the alignment rows to access position specific information. First, we have to define the type of the `Iterator`, which can be easily done by using the metafunction Iterator. Remember that we iterate over an `TRow` object. Then we have to construct the iterators `it` which points to the begin of `row1` using the begin() function and `itEnd` which points behind the last value of `row1` using the end() function. If you need to refresh the **Iterator Concept** you can read the iterator section *Iteration*. While we iterate over the gapped sequence, we can ask if the current value, at which the iterator `it` points to, is a gap or not by using the function isGap(). Use gapValue to print the correct gap symbol.

```
    typedef Iterator<TRow>::Type TRowIterator;
    TRowIterator it = begin(row1),
                 itEnd = end(row1);
    for (; it != itEnd; ++it)
```

```
    {
        TChar c = isGap(it) ? gapValue<TChar>() : *it;
        std::cout << c << " ";
    }
    std::cout << std::endl;
```

```
D - F G - -
```

We will now reset the clipping of `row1` using clearClipping and iterate again over it to see its effect.

```
    clearClipping(row1);

    it = begin(row1);
    itEnd = end(row1);
    for (; it != itEnd; ++it)
    {
        TChar c = isGap(it) ? gapValue<TChar>() : *it;
        std::cout << c << " ";
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

```
C D - F G - - D C
```

Here you can see how resetting the clipping positions brings back our complete row.

Assignment 1

Type Review

Objective Construct an alignment using the Align data structure for the sequences `"ACGTCACCTC"` and `"ACGGGCCTATC"`. Insert two gaps at the second position and insert one gap at the fifth position of the first sequence. Insert one gap at the ninth position of the second sequence. Iterate over the rows of your Align object and print the total count of gaps that exist within the alignment.

Hints You can use the function countGaps to count the number of consecutive gaps starting from the current position of the iterator.

The resulting alignment should look like:

```
AC--GTC-ACCTC

ACGGGCCTA--TC
```

Solution

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    // Defining all types that are needed.
```

```
    typedef String<char> TSequence;
    typedef Align<TSequence, ArrayGaps> TAlign;
    typedef Row<TAlign>::Type TRow;
    typedef Iterator<TRow>::Type TRowIterator;

    TSequence seq1 = "ACGTCACCTC";
    TSequence seq2 = "ACGGGCCTATC";

    // Initializing the align object.
    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);

    // Use references to the rows of align.
    TRow & row1 = row(align, 0);
    TRow & row2 = row(align, 1);

    // Insert gaps.
    insertGaps(row1, 2, 2);
    insertGap(row1, 7);  // We need to pass the view position which is changed␣
↪due to the previous insertion.
    insertGaps(row2, 9, 2);

    // Initialize the row iterators.
    TRowIterator itRow1 = begin(row1);
    TRowIterator itEndRow1 = end(row1);
    TRowIterator itRow2 = begin(row2);

    // Iterate over both rows simultaneously.
    int gapCount = 0;
    for (; itRow1 != itEndRow1; ++itRow1, ++itRow2)
    {
        if (isGap(itRow1))
        {
            gapCount += countGaps(itRow1);  // Count the number of consecutive␣
↪gaps from the current position in row1.
            itRow1 += countGaps(itRow1);    // Jump to next position to check for␣
↪gaps.
            itRow2 += countGaps(itRow1);    // Jump to next position to check for␣
↪gaps.
        }
        if (isGap(itRow2))
        {
            gapCount += countGaps(itRow2);  // Count the number of consecutive␣
↪gaps from the current position in row2.
            itRow1 += countGaps(itRow2);    // Jump to next position to check for␣
↪gaps.
            itRow2 += countGaps(itRow2);    // Jump to next position to check for␣
↪gaps.
        }
    }
    // Print the result.
    std::cout << "Number of gaps: " << gapCount << std::endl;
}
```

```
Number of gaps: 5
```

ToC

Contents

Alignment Representation (Graph)

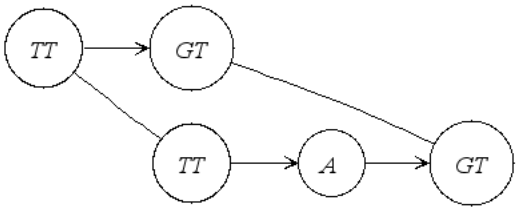
Learning Objective This tutorial introduces you to the graph data structures that can be used to represent an alignment in SeqAn. You will learn basic techniques to create and modify such data structures and how to access certain information from these data structures.

Difficulty Basic

Duration 15 min

Prerequisites *A First Example*, *Sequences*

Another very useful representation of alignments is given by the Alignment Graph. It is a graph in which each vertex corresponds to a sequence segment, and each edge indicates an ungapped alignment between the connected vertices, or more precisely between the sequences stored in those vertices. Here is an example of such a graph:



In the following we will actually construct this example step by step. First we include the `iostream` header from the STL and the `<seqan/align.h>` header to include all necessary functions and data structures we want to use. We use the namespace `seqan` and write the `main` function with an empty body.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
```

At the begin of the function we define our types we want to use later on. We define `TSequence` as the type of our input strings. Since we work with a Dna alphabet we define `TSequence` as a String over a Dna alphabet. For the AlignmentGraph we need two StringSets. The `TStringSet` is used to actually store the input sequences and the `TDepStringSet` is internally used by the AlignmentGraph. That is the AlignmentGraph does not copy the sources into its data structure but rather stores a reference to each of the given input strings as it does not modify the input sequences. The Dependent StringSet facilitates this behavior. In the end we define the actual AlignmentGraph type.

```
typedef String<Dna> TSequence;
typedef StringSet<TSequence> TStringSet;
typedef StringSet<TSequence, Dependent<> > TDepStringSet;
typedef Graph<Alignment<TDepStringSet> > TAlignGraph;
typedef typename VertexDescriptor<TAlignGraph>::Type TVertexDescriptor;
```

We first create our two input sequences `TTGT` and `TTAGT` append them to the StringSet `strings` using the appendValue function and pass the initialized `strings` object as a parameter to the constructor of the AlignmentGraph `alignG`.

```
TSequence seq1 = "TTGT";
TSequence seq2 = "TTAGT";

TStringSet strings;
appendValue(strings, seq1);
appendValue(strings, seq2);

TAlignGraph alignG(strings);
std::cout << alignG << std::endl;
```

Before adding vertices to the graph `align` prints the empty adjacency and edge list.

```
Adjacency list:
Edge list:
```

Before we construct the alignment we print the unmodified AlignmentGraph. Then we add some alignment information to the graph. In order to add an ungapped alignment segment we have to add an edge connecting two vertices of different input sequences. To do so we can use the function addEdge and specify the two vertices that should be connected. Since we do not have any vertices yet, we create them on the fly using the function addVertex(). The function addVertex gets as second parameter the id which points to the the correct input sequence within the `strings` object. We can use the function positionToId() to receive the id that corresponds to a certain position within the underlying Dependent StringSet of the AlignmentGraph.

We can access the Dependent StringSet using the function stringSet(). The third parameter of addVertex specifies the begin position of the segment within the respective input sequence and the fourth parameter specifies its length. Now, we add an edge between the two vertices of each input sequence which covers the first two positions. In the next step we have to add a gap. We can do this simply by just adding a vertex that covers the inserted string. Finally we have to add the second edge to represent the last ungapped sequence and then we print the constructed alignment.

Note that we use findVertex() to find the the last two inserted vertices. The syntax is the same as addVertex(), but omits the length parameter.

```
TVertexDescriptor u,v;

// TT
u = addVertex(alignG, positionToId(stringSet(alignG), 0), 0, 2);
v = addVertex(alignG, positionToId(stringSet(alignG), 1), 0, 2);
addEdge(alignG, u, v);

// A
addVertex(alignG, positionToId(stringSet(alignG), 1), 2, 1);

// GT
addVertex(alignG, positionToId(stringSet(alignG), 0), 2, 2);
addVertex(alignG, positionToId(stringSet(alignG), 1), 3, 2);

u = findVertex(alignG, positionToId(stringSet(alignG), 0), 2);
```

```
    v = findVertex(alignG, positionToId(stringSet(alignG), 1), 3);
    addEdge(alignG, u, v);

    std::cout << alignG << std::endl;

    return 0;
}
```

Now `align` prints the desired alignment.

```
Alignment matrix:
      0     .
        TT-GT
        || ||
        TTAGT
```

The general usage of graphs is explained in the *Graphs* tutorial.

Assignment 1

Type Review

Objective Construct a multiple sequence alignment using the Alignment Graph data structure. Use the three sequences `GARFIELDTHECAT`, `GARFIELDTHEBIGCAT` and `THEBIGCAT` and align them such that you obtain the maximal number of matches.

Hints `TSequence` should be `String<char>` instead of `String<Dna>`.

The function findVertex returns the vertex of an AlignmentGraph that covers the given position in the given sequence.

Solution

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    // Define the types we need.
    typedef String<char> TSequence;
    typedef StringSet<TSequence> TStringSet;
    typedef StringSet<TSequence, Dependent<> > TDepStringSet;
    typedef Graph<Alignment<TDepStringSet> > TAlignGraph;
    typedef typename VertexDescriptor<TAlignGraph>::Type TVertexDescriptor;

    // Initializing the sequences and the string set.
    TSequence seq1 = "GARFIELDTHECAT";
    TSequence seq2 = "GARFIELDTHEBIGCAT";
    TSequence seq3 = "THEBIGCAT";

    TStringSet strings;
    appendValue(strings, seq1);
    appendValue(strings, seq2);
    appendValue(strings, seq3);

    // Load the string set into the Alignment Graph.
```

```
    TAlignGraph alignG(strings);
    TVertexDescriptor u,v;

    // Add two vertices covering "GARFIELD" in the first and the second sequence␣
↪and connect them with an edge.
    u = addVertex(alignG, positionToId(stringSet(alignG), 0), 0, 8);
    v = addVertex(alignG, positionToId(stringSet(alignG), 1), 0, 8);
    addEdge(alignG, u, v);

    // Add two vertices covering "THE" in the first and the second sequence and␣
↪connect them with an edge.
    u = addVertex(alignG, positionToId(stringSet(alignG), 0), 8, 3);
    v = addVertex(alignG, positionToId(stringSet(alignG), 1), 8, 3);
    addEdge(alignG, u, v);

    // Find the vertex covering "THE" in the first sequence and add the vertex␣
↪covering "THE" in the third sequence and connect them with an edge.
    u = findVertex(alignG, positionToId(stringSet(alignG), 0), 8);
    v = addVertex(alignG, positionToId(stringSet(alignG), 2), 0, 3);
    addEdge(alignG, u, v);

    // Find the vertices covering "THE" in the second and the third sequence and␣
↪connect them with an edge.
    u = findVertex(alignG, positionToId(stringSet(alignG), 1), 8);
    v = findVertex(alignG, positionToId(stringSet(alignG), 2), 0);
    addEdge(alignG, u, v);

    // Add two vertices covering "FAT" in the second and the third sequence and␣
↪connect them with an edge.
    u = addVertex(alignG, positionToId(stringSet(alignG), 1), 11, 3);
    v = addVertex(alignG, positionToId(stringSet(alignG), 2), 3, 3);
    addEdge(alignG, u, v);

    // Add two vertices covering "CAT" in the first and the second sequence and␣
↪connect them with an edge.
    u = addVertex(alignG, positionToId(stringSet(alignG), 0), 11, 3);
    v = addVertex(alignG, positionToId(stringSet(alignG), 1), 14, 3);
    addEdge(alignG, u, v);

    // Find the vertex covering "CAT" in the first sequence and add the vertex␣
↪covering "CAT" in the third sequence and connect them with an edge.
    u = findVertex(alignG, positionToId(stringSet(alignG), 0), 11);
    v = addVertex(alignG, positionToId(stringSet(alignG), 2), 6, 3);
    addEdge(alignG, u, v);

    // Find the vertices covering "CAT" in the second and the third sequence and␣
↪connect them with an edge.
    u = findVertex(alignG, positionToId(stringSet(alignG), 1), 14);
    v = findVertex(alignG, positionToId(stringSet(alignG), 2), 6);
    addEdge(alignG, u, v);

    std::cout << alignG << std::endl;

    return 0;
}
```

```
Alignment matrix:
      0     .     :     .
```

```
        GARFIELDTHE---CAT
        ||||||||||    |||
        GARFIELDTHEBIGCAT
                ||||||||
        --------THEBIGCAT
```

Alignment Algorithms (e.g. *Pairwise* and *Multiple*) are one of the core algorithms in SeqAn. In this section you can learn how SeqAn represents alignments as C++ Objects and how you could use those data structures for your own alignment algorithm. Furthermore, you can learn which different kinds of *Scoring Schemes* exist, i.e. which combinations of Match/Mismatch Evaluation (e.g. Simple Score, Substitutional Matrices Score) and Insertion/Deletion Evaluation (e.g. Linear Gap Model, Affine Gap Model and Dynamic Gap Model) are possible and how you can define your own Scoring Matrices.

Store

ToC

Contents

Genome Annotations

Learning Objective You will learn how to work with annotations in SeqAn. After this tutorial, you will be able to write your own programs using annotations and analyzing them. You will be ready to continue with the *Fragment Store* Tutorial, e.g. if you want to combine your annotations with information from alignments.

Difficulty Average

Duration 1 h

Prerequisites *Sequences*

This tutorial will present SeqAn's efficient and easy-to-use data structures to work with annotations. They allow to annotate genome regions with features like 'gene', 'mRNA', 'exon', 'intron' and if required with custom features. We will give you an understanding of how to load annotations from a GFF or GTF file, store them in efficient data structures, as well as how to traverse and access these information.

AnnotationStore as Part of the FragmentStore

This section will give you a short introduction to data structures relevant for working with annotations.

In SeqAn, annotations are stored in the so-called annotationStore, which is part of the FragmentStore. The annotationStore can only be used together with the FragmentStore, because the latter stores additional information, e.g. the contig names or sequences. The FragmentStore is a data structure specifically designed for read mapping, genome assembly or gene annotation.

The FragmentStore can be seen as a database, where each table (called "store") is implemented as a String. Each row of the table corresponds to an element in the string. The position of each element in the string implicitly represents the Id of such element in the table. All such strings are members of the class FragmentStore, are always present and empty if unused. For example, the member contigStore is a string of elements, each one containing among others a contig sequence.

For detailed information about the FragmentStore read the *Fragment Store* Tutorial.

Accordingly, the annotationStore is a String, where each element represents one annotation. Each element holds the necessary information, e.g. beginPos, endPos, parentId etc., as data members.

AnnotationStore

In this section you will learn how to work with the annotationStore itself.

Annotations are represented hierarchically by a tree having at least a root node.

A typical annotation tree looks as follows.

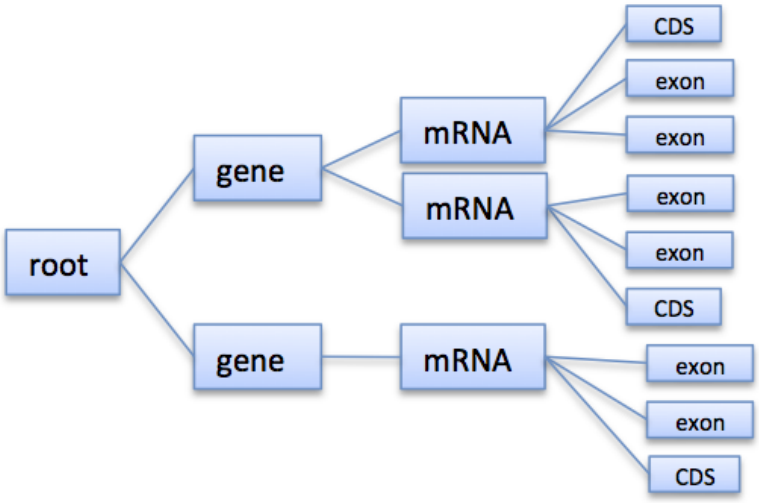


Fig. 4.3: Annotation tree example

The following entity-relationship diagram shows the tables holding store annotations, their relationships and cardinalities.

The instantiation of an annotationStore happens implicitly with the instantiation of a FragmentStore. To access the FragmentStore definitions we'll need to include the correct header:

```
#include <seqan/store.h>
```

Now we can simply write:

```
    FragmentStore<> store;
```

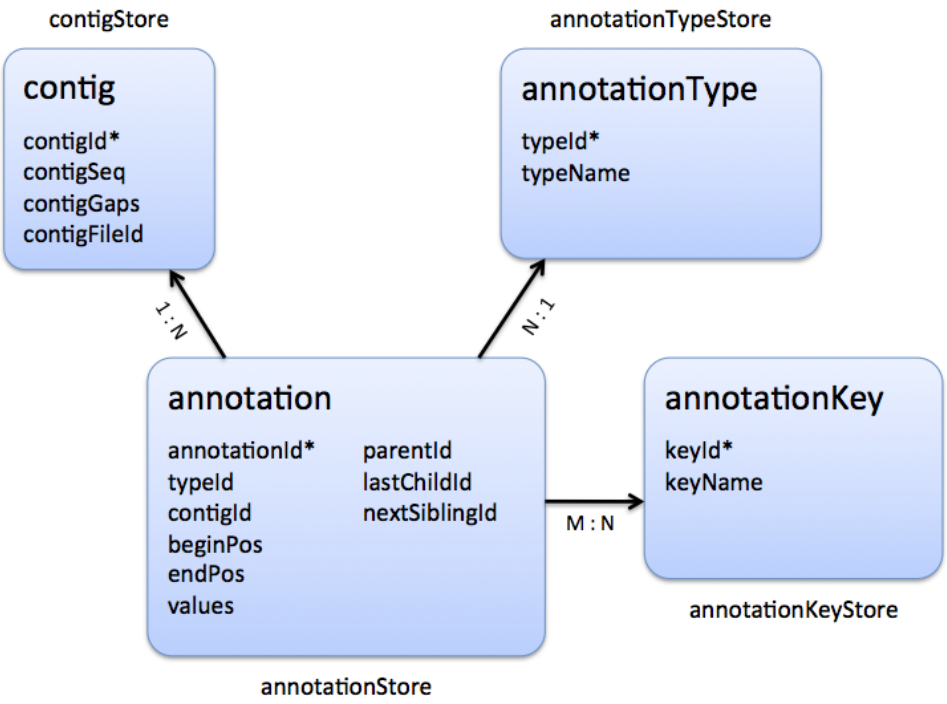


Fig. 4.4: Stores involved in gene annotation

Loading an Annotation File

Before we deal with the actual annotation tree, we will first describe how you can easily load annotations from a GFF or GTF file into the FragmentStore.

An annotation file can be read from an GffFileIn with the function readRecords. The file extension specifies if we want to read a GFF, GTF or UCSC file. The following example shows how to read an GTF file:

```
    CharString fileName = getAbsolutePath("demos/tutorial/genome_annotations/example.
↪gtf");
    GffFileIn file(toCString(fileName));

    readRecords(store, file);
```

The GFF-reader is also able to detect and read GTF files. The UCSC Genome Browser uses two seperate files, the `kownGene.txt` and `knownIsoforms.txt`. They must be read by using two different UcscFileIn objects (one for `knownGene.txt` and one for `knownIsoforms.txt`). Finally you call readRecords with both UcscFileIn objects.

Tip: An annotation can be loaded without loading the corresponding contigs.

In that case empty contigs are created in the contigStore with names given in the annonation. A subsequent call of loadContigs would load the sequences of these contigs, if they have the same identifier in the contig file.

Traversing the Annotation Tree

This section will illustrate how to use iterators to traverse the annotation tree.

The annotation tree can be traversed and accessed with the AnnotationTree Iterator. Again we use the metafunction *dox:ContainerConcept#Iterator Iterator* to determine the appropriate iterator type for our container. A new AnnotationTree iterator can be obtained by calling begin with a reference to the FragmentStore and the `AnnotationTree` tag:

```
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
    it = begin(store, AnnotationTree<>());
```

The AnnotationTree iterator starts at the root node and can be moved to adjacent tree nodes with the functions goDown, goUp, and goRight. These functions return a boolean value that indicates whether the iterator could be moved. The functions isLeaf, isRoot, isLastChild return the same boolean without moving the iterator. With goRoot or goTo the iterator can be moved to the root node or an arbitrary node given its annotationId. If the iterator should not be moved but a new iterator at an adjacent node is required, the functions nodeDown, nodeUp, nodeRight can be used.

```
    // Move the iterator down to a leaf
    while (goDown(it))
    {}
    // Create a new iterator and if possible move it to the right sibling of the␣
↪first iterator
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it2;
    if (isLastChild(it))
        it2 = nodeRight(it);
```

The AnnotationTree iterator supports a preorder DFS traversal and therefore can also be used in typical begin-end loops with the functions goBegin (== goRoot), goEnd, goNext, atBegin, atEnd. During a preorder DFS, the descent into subtree can be skipped by goNextRight, or goNextUp which proceeds with the next sibling or returns to the parent node and proceeds with the next node in preorder DFS.

```
    // Move the iterator back to the beginning
    goBegin(it);
    // Iterate over the nodes in preorder DFS while the end is not reached and
    // output if the current node is a leaf
    while (!atEnd(it))
    {
        if (isLeaf(it))
            std::cout << " current node is leaf" << std::endl;
        goNext(it);
    }
```

Asignment 1

Type Review

Objective Copy the code below, which loads the annotations from a given GTF file into the FragmentStore and initializes an iterator on the AnnotationTree. Download the GTF file `assignment_annotations.gtf`, whose annotations build an AnnotationTree of the typical structure with gene, mRNA and exon level. Adjust the code to go down to the exon level and iteratate over all children of the first mRNA and count them. Print the result.

Click **more...** to see the code.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/store.h>

using namespace seqan;

int main()
{
    CharString fileName = getAbsolutePath("demos/tutorial/genome_annotations/
↪assignment_annotations.gtf");
    GffFileIn file(toCString(fileName));

    FragmentStore<> store;
    readRecords(store, file);
    // Create AnnotationTree iterator
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
    it = begin(store, AnnotationTree<>());
    // Move iterator one node down
    goDown(it);

    std::cout << "Is leaf: " << isLeaf(it) << std::endl;
    return 0;
}
```

Hints In the given data the left-most leaf is a child of mRNA and has siblings. You can use the function goRight to traverse over all siblings.

Solution Click **more...** to see one possible solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/store.h>

using namespace seqan;
int main()
{
    CharString fileName = getAbsolutePath("demos/tutorial/genome_annotations/
↪assignment_annotations.gtf");
    GffFileIn file(toCString(fileName));

    FragmentStore<> store;
    readRecords(store, file);
    // Create AnnotationTree iterator
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
    it = begin(store, AnnotationTree<>());
    unsigned count = 0;
    // Go down to the first leaf (first child of the first mRNA)
    while (goDown(it))
    {}
    std::cout << "Is leaf: " << isLeaf(it) << std::endl;

    ++count;
    // Iterate over all siblings and count
    while (goRight(it))
        ++count;
    std::cout << "No. of children of the first mRNA: " << count << std::endl;
```

```
    return 0;
}
```

```
Is leaf: 1
No. of children of the first mRNA: 9
```

Assignment 2

Type Review

Objective Reuse the code and the GTF file from above. Instead of counting only the children of the first mRNA adjust the code to count the children for each given mRNA. Print the results.

Hints After you reached the last child of the first mRNA you can use the functions goNext and goDown to traverse to the next leaf.

Solution Click **more...** to see one possible solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/store.h>

using namespace seqan;

int main()
{
    CharString fileName = getAbsolutePath("demos/tutorial/genome_annotations/
↪assignment_annotations.gtf");
    GffFileIn file(toCString(fileName));

    FragmentStore<> store;
    readRecords(store, file);
    // Iterate over all leafs, count and print the result
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
    it = begin(store, AnnotationTree<>());
    unsigned count = 0;
    std::cout << "Number of children for each mRNA: " << std::endl;
    // Go down to the first leaf (first child of the first mRNA)
    while (goDown(it))
    {}

    while (!atEnd(it))
    {
        ++count;
        // Iterate over all siblings and count
        while (goRight(it))
            ++count;
        std::cout << count << std::endl;
        count = 0;
        // Jump to the next mRNA or gene, go down to its first leaf and count it
        if (!atEnd(it))
        {
            goNext(it);
            if (!atEnd(it))
                while (goDown(it))
                {}
```

```
        }
    }
    return 0;
}
```

```
Number of children for each mRNA:
9
2
2
```

Accessing the Annotation Tree

Let us now have a closer look how to access the information stored in the different stores representing the annotation tree.

To access or modify the node an iterator points at, the iterator returns the node's annotationId by the value function (== `operator*`). With the annotationId the corresponding entry in the annotationStore could be modified manually or by using convenience functions. The function getAnnotation returns a reference to the corresponding entry in the annotationStore. getName and setName can be used to retrieve or change the identifier of the annotation element. As some annotation file formats don't give every annotation a name, the function getUniqueName returns the name if non-empty or generates one using the type and id. The name of the parent node in the tree can be determined with getParentName. The name of the annotation type, e.g. 'mRNA' or 'exon', can be determined and modified with getType and setType.

Assume we have loaded the file `example.gtf` with the following content to the FragmentStore *store* and instantiated the iterator *it* of the corresponding annotation tree.

```
chr1        MySource        exon        150        200        .        +        .
↪        gene_id "381.000"; transcript_id "381.000.1";
chr1        MySource        exon        300        401        .        +        .
↪        gene_id "381.000"; transcript_id "381.000.1";
chr1        MySource        CDS        380        401        .
↪        +        0        gene_id "381.000"; transcript_id "381.000.1";
chr1        MySource        exon        160        210        .        +        .
↪        gene_id "381.000"; transcript_id "381.000.2";
```

We now want to iterate to the first exon and output a few information:

```
    // Move the iterator to the begin of the annotation tree
    it = begin(store, AnnotationTree<>());
    // Go down to exon level
    while (goDown(it)) ;
    std::cout << "type: " <<  getType(it) << std::endl;
    std::cout << "id: " << value(it) << std::endl;
    std::cout << "begin position: " <<  getAnnotation(it).beginPos << std::endl;
```

For our example the output would be:

```
type: exon
id: 3
begin position: 149
```

An annotation can not only refer to a region of a contig but also contain additional information given as key-value pairs. The value of a key can be retrieved or set by getValueByKey and assignValueByKey. The values of a node can be cleared with clearValues.

A new node can be created as first child, last child, or right sibling of the current node with createLeftChild, createRightChild, or createSibling. All three functions return an iterator to the newly created node.

```
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it3;
    // Create a right sibling of the current node and return an iterator to this new␣
↪node
    it3 = createSibling(it);
```

The following list summarizes the functions provided by the AnnotationTree iterator.

getAnnotation, value Return annotation object/id of current node

getName, setName, getType, setType Access name or type of current annotation object

getParentName Access parent name of current annotation object

clearValue, getValueByKey, assignValueByKey Access associated values

goBegin, goEnd, atBegin, atEnd Go to or test for begin/end of DFS traversal

goNext, goNextRight, goNextUp go next, skip subtree or siblings during DFS traversal

goRoot, goUp, goDown, goRight Navigate through annotation tree

createLeftChild, createRightChild, createSibling Create new annotation nodes

isRoot, isLeaf Test for root/leaf node

Assignment 3

Type Application

Objective Again use the given GTF file `assignment_annotations.gtf` and create an iterator on the annotation tree. Now iterate to the first node of type "exon" and output the following features:

1. type
2. begin position
3. end position
4. its Id
5. the Id of its parent
6. the name of its parent

Solution Click **more...** to see one possible solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/store.h>

using namespace seqan;

int main()
{
    CharString fileName = getAbsolutePath("demos/tutorial/genome_annotations/
↪assignment_annotations.gtf");
    GffFileIn file(toCString(fileName));

    FragmentStore<> store;
```

```
    readRecords(store, file);
    // Create iterator
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
    it = begin(store, AnnotationTree<>());
    // Iterate to the first annotation of type "exon"
    while (!atEnd(it) && getType(it) != "exon")
        goNext(it);
    // Output:
    std::cout << "  type: " << getType(it) << std::endl;
    std::cout << "  begin position: " << getAnnotation(it).beginPos << std::endl;
    std::cout << "  end position: " << getAnnotation(it).endPos << std::endl;
    std::cout << "  id: " << value(it) << std::endl;
    std::cout << "  parent id: " << getAnnotation(it).parentId << std::endl;
    std::cout << "  parent name: " << getParentName(it) << std::endl;
    return 0;
}
```

```
  type: exon
  begin position: 149
  end position: 200
  id: 3
  parent id: 2
  parent name: 381.000.1
```

Assignment 4

Objective Write a small statistic tool to analyse a given set of annotations.

1. Load the annotations given in the GTF file `assignment_annotations.gtf`.
2. Output the average number of mRNAs for genes.
3. Output the average number of exons for mRNAs.
4. Additionally output the average exon length.
5. Test your program also on large data, e.g. the annotation of the mouse genome Mus_musculus.NCBIM37.61.gtf.zip (don't forget to unzip first).

Solution Click **more...** to see one possible solution.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/store.h>

using namespace seqan;

int main()
{
    CharString fileName = getAbsolutePath("demos/tutorial/genome_annotations/
↪assignment_annotations.gtf");
    GffFileIn file(toCString(fileName));

    FragmentStore<> store;
    readRecords(store, file);
    // Create iterator
    Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
```

```
    it = begin(store, AnnotationTree<>());
    unsigned countGenes = 0;
    unsigned countmRNAs = 0;
    unsigned countExons = 0;
    unsigned length = 0;
    // Iterate over annotation tree and count different elements and compute exon␣
↪lengths
    while (!atEnd(it))
    {
        if (getType(it) == "gene")
        {
            ++countGenes;
        }
        else if (getType(it) == "mRNA")
        {
            ++countmRNAs;
        }
        else if (getType(it) == "exon")
        {
            ++countExons;
            length += abs((int)getAnnotation(it).endPos - (int)getAnnotation(it).
↪beginPos);
        }
        goNext(it);
    }
    if (countGenes == 0u)  // prevent div-by-zero below
        countGenes = 1;
    if (countmRNAs == 0u)  // prevent div-by-zero below
        countmRNAs = 1;
    if (countExons == 0u)  // prevent div-by-zero below
        countExons = 1;
    // Ouput some stats:
    std::cout << "Average number of mRNAs for genes: " << (float)countmRNAs /␣
↪(float)countGenes << std::endl;
    std::cout << "Average number of exons for mRNAs: " << (float)countExons /␣
↪(float)countmRNAs << std::endl;
    std::cout << "Average length of exons: " << (float)length / (float)countExons
↪<< std::endl;
    return 0;
}
```

```
Average number of mRNAs for genes: 1.5
Average number of exons for mRNAs: 3
Average length of exons: 95.5556
```

Write an Annotation File

To write an annotation to a file use the function writeRecords. Note that the format (`Gff()` or `Gtf()`) is specified by the file extension.

```
    // Open output stream
    GffFileOut fileOut("example_out.gtf");
    // Write annotations to GTF file
    writeRecords(fileOut, store);
```

ToC

Contents

Fragment Store

Learning Objective You will learn about the SeqAn FragmentStore for handling fragments. The "fragments" are reads and the data structure is useful in the context of read mapping, genome assembly, and gene annotation. After completing this tutorial, you will be able to use the most relevant functionality of the FragmentStore class.

Difficulty Advanced

Duration 1 h

Prerequisites *Getting Started*, *Sequences*

Overview

The FragmentStore is a data structure specifically designed for read mapping, genome assembly or gene annotation. These tasks typically require lots of data structures that are related to each other like:

- reads, mate-pairs, reference genome
- pairwise alignments
- genome annotation

The Fragment Store subsumes all these data structures in an easy to use interface. It represents a multiple alignment of millions of reads or mate-pairs against a reference genome consisting of multiple contigs. Additionally, regions of the reference genome can be annotated with features like 'gene', 'mRNA', 'exon', 'intron' or custom features. The Fragment Store supports I/O functions to read/write a read alignment in SAM/BAM or AMOS format and to read/write annotations in GFF or GTF format.

The Fragment Store can be compared with a database where each table (called "store") is implemented as a String member of the FragmentStore class. The rows of each table (implemented as structs) are referred by their ids which are

their positions in the string and not stored explicitly (marked with $\star$ in the Figures 2 and 5). The only exception is the alignedReadStore whose elements of type AlignedReadStoreElement contain an id-member as they may be rearranged in arbitrary order, e.g. by increasing genomic positions or by readId. Many stores have an associated name store to store element names. Each name store is a StringSet that stores the element name at the position of its id. All stores are present in the Fragment Store and empty if unused. The concrete types, e.g. the position types or read/contig alphabet, can be easily changed by defining a custom config struct which is a template parameter of the Fragment Store class.

Multiple Read Alignment

The Fragment Store can represent a multiple read alignment, i.e. is an alignment between the contigs and the set of reads, where one read can be aligned at zero, one or multiple positions of a contig. In the multiple alignment the contig is represented by one line with gaps (-) and the remaining lines are to reads or read segments with gaps aligned to the contig. The following figure shows one contig (the line at the top) and multiple reads aligned to it arranged as stairs (reads in lower-case align to the reverse strand):

```
TGAAAACTATATTTATGCTATTCAGTTCTAAATATAGAAATTGAAACAGCTGTGTTTAGTGCCTTTGTTCA-----
 ↪ACCCCCTTGCAACAACCTTGAGAACCCCAGGGAATTTGTCAATGTCAGGGAAGGAGCATTTTGTCAGTTACCAAATGTGTTTATTACCAG
TGAAAACTATATT ATGCTATTCAGTTCTAAATATAGAAATTGAAACAG  GTGTTTAGTGCCTTTGTTCA-----
 ↪ACCCCCTTGCAACAAC      aaccccagggaatttgtcaatgtcagggaaggagc⌴
 ↪ttttgtcagttaccaaatgtgtttattaccag
tgaa  ctatatttatgctattcagttctaaatatagaaatt   acagctgtgtttagtgcctttgttca-----acccccttg⌴
 ↪aacaaccttgagaaccccagggaatttgtcaatgt   GGAAGGAGCATTTTGTCAGTTACCAAATGTGTTT  TACCAG
TGAAAACTATAT TATGCTATTCAGTTCTAAATATAGAAATTGAAACA ctgtgtttagtgcctttgttca-----
 ↪acccccttgcaac ACCTTGAGAACCCCAGGGAATTTGTCAATGTCAGG ⌴
 ↪aggagcattttgtcagttaccaaatgtgtttatta  at
TGAAAACTATATTTA gctattcagttctaaatatagaaattgaaacagct ⌴
 ↪GTTTAGTGCCTTTGTTCACATAGACCCCCTTGCAA aaccttgagaaccccagggaatttgtcaatgtcag  ⌴
 ↪aggagcattttgtcagttaccaaatgtgtttatta  AG
TGAAAACTATATTTATGCTATTCAGT          GAAATTGAAACAGCTGTGTTTAGTGCCTTTGTTCA      ⌴
 ↪ccccttacaacaaccttgagaaccccagggaattt        CAGGGAAGGAGCATTTTGTCAGTTACCAAATGTGT     ⌴
 ↪   G
tgaaaactatatttatgctattcagt                                  ⌴
 ↪GCCTTTGTTCACATAGACCCCCTTGCAACAACCTT        cagggaatttgtcaatgtcagggaaggagcatttt ⌴
 ↪CAGTTACCAAATGTGTTTATTACCAG
tgaaaactatatttatgctattcagttcta                                  AG-----
 ↪ACCCCCTTGCAACAACCTTGAGAACCCCAGGGA              ggaaggagcattttgtcagttaccaaatgtgttta
TGAAAACTATATTTATGCTATTCAGTTCTAA                                  A-----
 ↪ACCCCCTTGCAACAACCTTGAGAACCCCAGGGAA             gaaaggagcattttgtcagttaccaaatgtgttta
TGAAAACTATATTTATGCTATTCAGTTCTAAA                                 A-----
 ↪ACCCCCTTGCAACAACCTTGAGAACCCCAGGGAA               ⌴
 ↪AGGAGCATTTTGTCAGTTACCAAATGTGTTTATTA
TGAAAACTATATTTATGCTATTCAGTTCTAAA                                                ⌴
 ↪TGCAACAACCTTGAGAACCCCAGGGAATTTGTCAA          ggagcattttgtcagttaccaaatgtgtttattac
TGAAAACTATATTTATGCTATTCAGTTCTAAAT                                               ⌴
 ↪TGCAACAACCTTGAGAACCCCAGGGAATTTGTCAA          GGAGCATTTTGTCAGTTACCAAATGTGTTTATTAC
TGAAAACTATATTTATGCTATTCAGTTCTAAAT                                               ⌴
 ↪TGCAACAACCTTGAGAACCCCAGGGAATTTGTCAA          GGAGCATTTTGTCAGTTACCAAATGTGTTTATTAT
      ctatatttatgctattcagttctaaatatagaaatt                                      ⌴
 ↪tgcaacaaccttgagaaccccagggaatttgtcaa          GGAGCATTTTGTCAGTTACCAAATGTGTTTATTAC
      ctatatttatgctattcagttctaaatatagaaatt                                          ⌴
 ↪  CAACCTTGAGAACCCCAGGGAATTTGTCAATGTCA       agcattttgtcagttaccaaatgtgtttattacca
         TATTTATGCTATTCAGTTATAAATATAGAAATTGAAACAG                                   ⌴ ⌴
 ↪     CCTTGAGAACCCCAGGGAATTTGTCAATGTCAGGG    agcattttgtcagttaccaaatgtgtttattacca
           atttatgctattcagttctaaatatagaaattgaa                                   ⌴ ⌴
 ↪      CTTGAGAACCCCAGGGAATTTGTCAATGTCAGGGA    GCATTTTGTCAGTTACCAAATGTGTTTATTACCAG
            tttacgctattcagtactaaatatagaaattgaaa                                 ⌴ ⌴
 ↪      CTTGAGAACCCCAGGGAATTTGTCAATGTCAGGGA    GCATTTTGTCAGTTACCAAATGTGTTTATTACCAG
```

```
            ttatgctattcagttctaaatatagaaattgaaac
↪                   gggaatttgtcaatgtcagggaaggagcattttgt AGTTACCAAATGTGTTTATTACCAG
```

- ***Figure 1:*** Multiple read alignment

The following figure shows which tables represent the multiple read alignment:

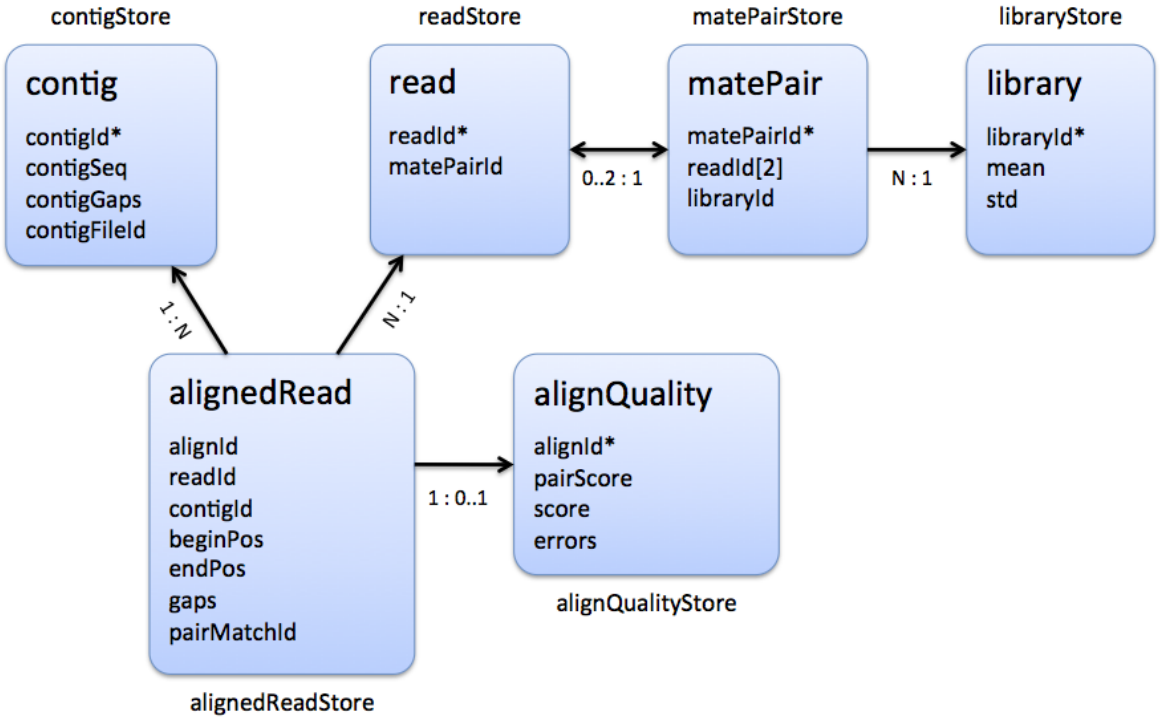


Fig. 4.5: ***Figure 2:*** Stores used to represent a multiple read alignment

The main table is the alignedReadStore which stores AlignedReadStoreElements. Each entry is an alignment of a read (`readId`) and a contig (`contigId`). Introduced gaps are stored as a string of gap anchors in the `gaps` member of the alignedReadStore entry and the contigStore entry. The begin and end positions of the alignment are given by the `beginPos` and `endPos` members which are 0-based positions on the forward strand in gap space, i.e. positions in the gapped contig sequence. If the read is aligned to the reverse strand it holds `endPos < beginPos`. However, the gaps are always related to the forward strand. Additional information, e.g. the number of errors, an alignment score or additional alignment tags, are stored in the tables alignQualityStore and alignedReadTagStore at position `id`, where `id` is a unique id of the AlignedReadStoreElement. Paired-end or mate pair alignments are represented by two entries in the alignedReadStore that have the same `pairMatchId` value (unequal to `INVALID_ID`). For orphaned read alignments holds `pairMatchId == INVALID_ID`.

```
       012345556789  sequence space
       012345678901       gap space
contig ACCAC--GTTTG
 read1   ACACGGT         [2-9[
 read2     ACGGTT-G      [4-12[
```

The alignedReadStore is the only store where the id (alignId in the figure) of an element is not implicitly given by its position. The reason for this is that it is necessary in many cases to rearrange the elements of the alignedReadStore, e.g. increasingly by (contigId,beginPos), by readId or pairMatchId. This can be done by sortAlignedReads. If it is necessary to address an element by its id, the elements must be sorted by id first. In the case that ids are not contiguously

increasing, e.g. because some elements where removed, they must be renamed by a prior call of compactAlignedReads. Analogously the function compactPairMatchIds renames `pairMatchId` values contiguously and replaces values that occur in only one alignment by `INVALID_ID`.

Display Aligned Reads

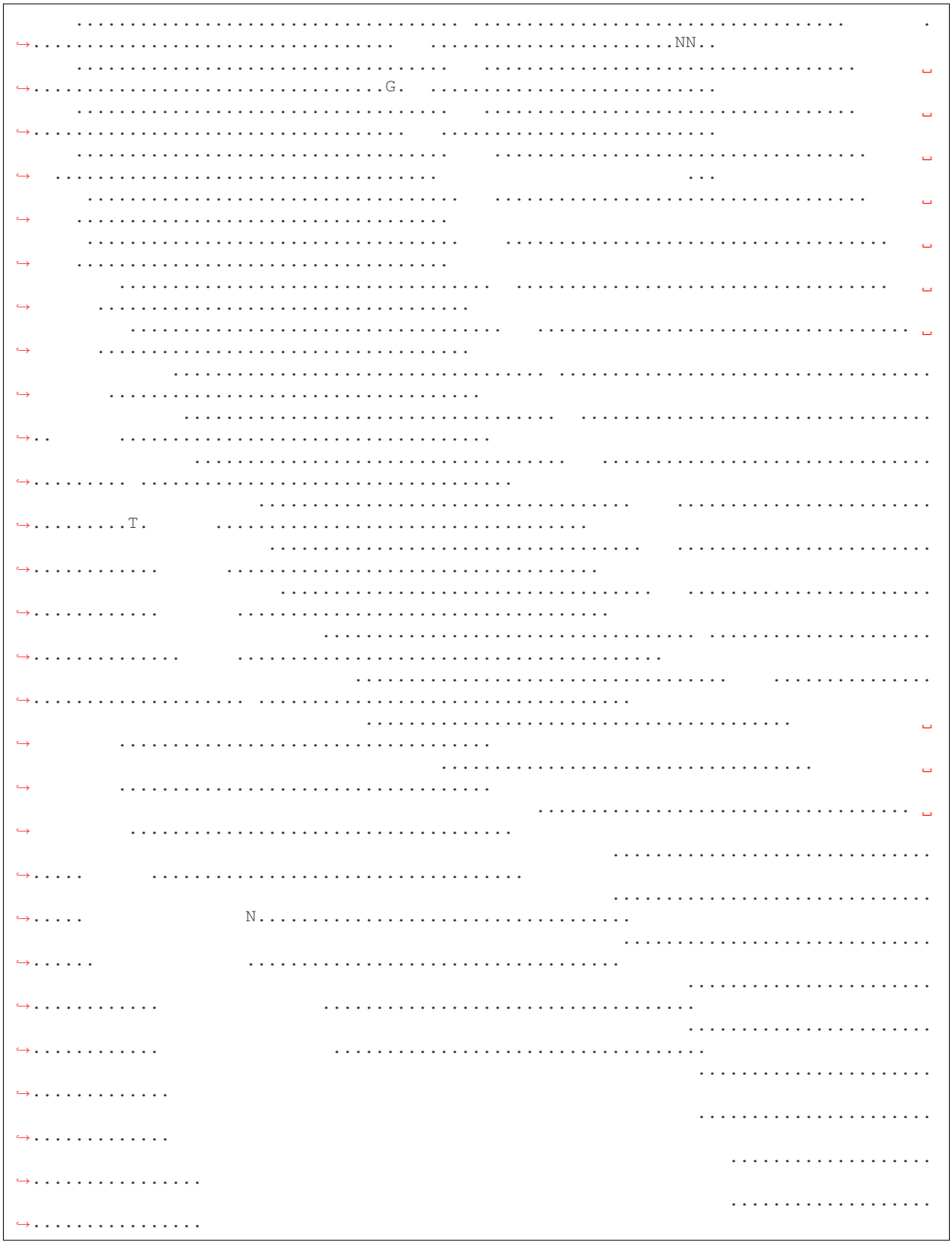
The multiple read alignment can be displayed in text form or in a scalable graphics format (SVG). Therefore first a stairs layout of the reads must be computed via layoutAlignment and stored in an AlignedReadLayout. The function printAlignment can then be used to output a window (beginPos,endPos,firstLine,lastLine) of the read alignment against a contig either to a stream or `SVGFile`. The following small example demonstrates how to first load two contigs from a Fasta file and then import a read alignment given in SAM format:

```
#include <iostream>
#include <seqan/store.h>
#include <seqan/misc/svg.h>

using namespace seqan;

int main()
{
    CharString fastaFileName = getAbsolutePath("demos/tutorial/fragment_store/example.
↪fa");
    CharString samFileName = getAbsolutePath("demos/tutorial/fragment_store/example.
↪sam");

    typedef FragmentStore<> TStore;

    TStore store;
    loadContigs(store, toCString(fastaFileName));
    BamFileIn file(toCString(samFileName));
    readRecords(store, file);
```

Then we create a stairs layout of the aligned reads and output a window from gapped position 0 to 150 and line 0 to 36 of the multiple alignments below contig 1 to standard out.

```
    AlignedReadLayout layout;
    layoutAlignment(layout, store);
    printAlignment(std::cout, layout, store, 1, 0, 150, 0, 36);
```

```
TTCAAATGAACTTCTGTAATTGAAAAATTCATTTAAGAAATTACAAAATATAGTTGAAAGCTCTAACAATAGACTAAACCAAGCAGAAGAAAGAGGTTCAG
..................................... ....................................   ..........
↪.......................... .................................... .
.................................... ................................... ..............
↪..................... ....................................    ...
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,, ....................................    ..........
↪..........C.............. .................................... .
 ....................................   .....................A.............    ........
↪.............................    ...............................
    .................................... ...................................     ......
↪..............................    ..............................
     ....................................  ...................................     ....
↪................................    ............A.................
     ....................................  ...................................      ...
↪.................................    ............................
     ....................................  ...................................      ...
↪..............N.NNN...N.........    ............................
```



The same window can also be exported as a scalable vector graphic in SVG format.

```
    SVGFile svg("layout.svg");
    printAlignment(svg, layout, store, 1, 0, 150, 0, 36);

    return 0;
}
```

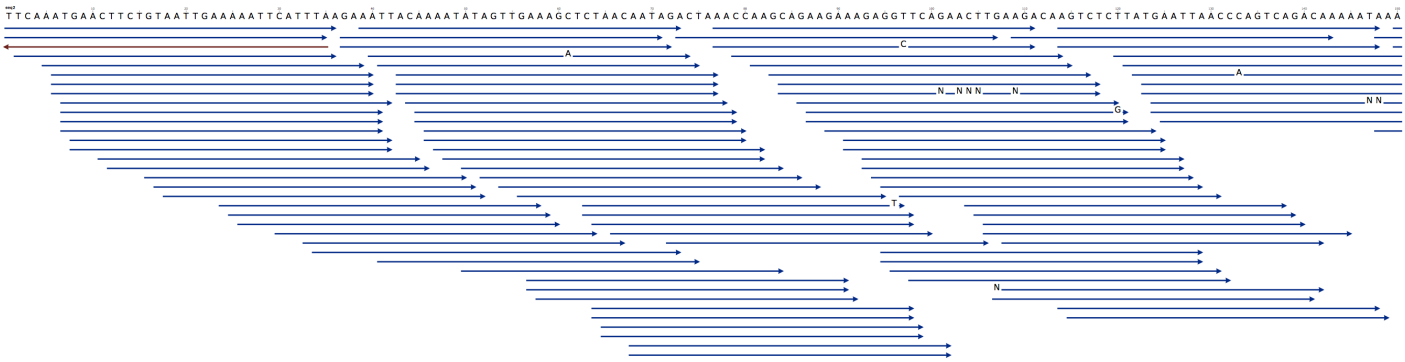


Fig. 4.6: ‘’‘Figure 3:’‘’ SVG export of a multiple read alignment

Accessing Pairwise Alignments

In the next step, we want to access several pairwise alignments between reads and contig segments. Therefore we first need to get the associated types that the Fragment Store uses to store contig and read sequences and gaps. This can be done by the following typedefs:

```
    typedef Value<TStore::TContigStore>::Type                                TContig;
    typedef Value<TStore::TAlignedReadStore>::Type
↪TAlignedRead;

    typedef Gaps<TContig::TContigSeq, AnchorGaps<TContig::TGapAnchors> >
↪TContigGaps;
    typedef Gaps<TStore::TReadSeq, AnchorGaps<TAlignedRead::TGapAnchors> >  TReadGaps;

    TStore::TReadSeq readSeq;
```

Now we want to extract and output the alignments from the alignedReadStore at position 140,144,...,156. First we store a reference of the alignedRead in ar as we need to access it multiple times. The read sequence is neither stored in the readStore or alignedReadStore as many short sequences can more efficiently be stored in a separate StringSet like the readSeqStore. We copy the read sequence into a local variable (defined outside the loop to save allocations/deallocations) as we need to compute the reverse-complement for reads that align to the reverse strand. Then we create a gaps object that represent the alignment rows of the contig and the aligned read in the multiple sequence alignment. The Gaps object requires references of the sequence and the gap-anchor string stored in the contigStore and the alignedReadStore. We need to limit the view of the contig alignment row to the interval the read aligns to, i.e. the gap position interval [beginPos,endPos[. After that we output both alignment rows.

Tip: The Gaps contains two Holder references to the sequence and the inserted gaps. In our example these Holders are dependent and changes made to the Gaps object like the insertion/deletion of gaps would immediatly be persistent in the Fragment Store.

```
    for (int i = 140; i < 160; i += 4)
    {
        TAlignedRead & ar = store.alignedReadStore[i];

        readSeq = store.readSeqStore[ar.readId];
        if (ar.endPos < ar.beginPos)
            reverseComplement(readSeq);

        TContigGaps contigGaps(
            store.contigStore[ar.contigId].seq,
            store.contigStore[ar.contigId].gaps);

        TReadGaps readGaps(
            readSeq,
            ar.gaps);

        setBeginPosition(contigGaps, std::min(ar.beginPos, ar.endPos));
        setEndPosition(contigGaps, std::max(ar.beginPos, ar.endPos));

        std::cout << "ALIGNMENT " << i << std::endl;
        std::cout << "\tcontig " << ar.contigId << ":\t" << contigGaps;
        std::cout << "     \t[" << beginPosition(contigGaps) << ".." <<␣
↪endPosition(contigGaps) << "[" << std::endl;
        std::cout << "\tread "   << ar.readId   << ":\t" << readGaps << std::endl;
        std::cout << std::endl;
    }
```

```
ALIGNMENT 140
        contig 0:       CTGTGTTTAGTGCCTTTGTTCA-----ACCCCCTTGCAACAACCT     ␣
↪         [266..306[
        read 149:       CTGTGTTTAGTGCCTTTGTTCA-----ACCCCCTTGCAAC

ALIGNMENT 144
        contig 0:       AGTGCCTTTGTTCA-----ACCCCCTTGCAACAACCTTGAG             [274..
↪310[
        read 153:       AGTGCCTTTGTTCACATAGACCCCCTTGCAACAACC

ALIGNMENT 148
        contig 0:       TTCA-----ACCCCCTTGCAACAACCTTGAGAACCCCAGGGAATT     ␣
↪         [284..324[
        read 157:       ATAG-----ACCCCCTTGCAACAACCTTGAGAACCCCAGG

ALIGNMENT 152
        contig 0:       CA-----ACCCCCTTGCAACAACCTTGAGAACCCCAGGGAATTTG     ␣
↪         [286..326[
        read 161:       CA-----ACCCCCTTGCAACAACCTTGCGAACCCCAGGGA

ALIGNMENT 156
        contig 0:       TTGCAACAACCTTGAGAACCCCAGGGAATTTGTCA             [294..329[
        read 165:       CCCCCTTGCAACAACCTTGAGAACCCCAGGGAATT
```

Assignment 1

Type Rview

Objective Modify the example above, such that reads that align to the reverse strand are displayed in lower-case

letters.

Difficulty Easy

Hint The Dna alphabet used in the fragment store doesn't support lower-case letters. You have to use a string of chars for readSeq.

Solution As we copy the read sequence, it suffices to change the type of the target string readSeq and the sequence type of the read Gaps object into CharString, i.e. a String of `char`.

```
    typedef Value<TStore::TContigStore>::Type
↪TContig;
    typedef Value<TStore::TAlignedReadStore>::Type
↪TAlignedRead;

    typedef Gaps<TContig::TContigSeq, AnchorGaps<TContig::TGapAnchors> >
↪TContigGaps;
    typedef Gaps<TStore::TReadSeq, AnchorGaps<TAlignedRead::TGapAnchors> >
↪TReadGaps;

    CharString readSeq;
```

Then, we not only need to reverse-complement readSeq if the read aligns to the reverse strand (`endPos < beginPos`) but also need to convert its letters into lower-case. Therefor SeqAn provides the function toLower. Alternatively, we could iterate over readSeq and add ('a'-'A') to its elements.

```
    for (int i = 140; i < 160; i += 4)
    {
        TAlignedRead & ar = store.alignedReadStore[i];

        readSeq = store.readSeqStore[ar.readId];
        if (ar.endPos < ar.beginPos)
        {
            reverseComplement(readSeq);
            toLower(readSeq);
        }

        TContigGaps contigGaps(
            store.contigStore[ar.contigId].seq,
            store.contigStore[ar.contigId].gaps);

        TReadGaps readGaps(
            readSeq,
            ar.gaps);

        setBeginPosition(contigGaps, std::min(ar.beginPos, ar.endPos));
        setEndPosition(contigGaps, std::max(ar.beginPos, ar.endPos));

        std::cout << "ALIGNMENT " << i << std::endl;
        std::cout << "\tcontig " << ar.contigId << ":\t" << contigGaps;
        std::cout << "     \t[" << beginPosition(contigGaps) << ".." <<
↪endPosition(contigGaps) << "[" << std::endl;
        std::cout << "\tread "   << ar.readId   << ":\t" << readGaps << std::endl;
        std::cout << std::endl;
    }
```

Running this program results in the following output.

```
ALIGNMENT 140
        contig 0:        CTGTGTTTAGTGCCTTTGTTCA-----ACCCCCTTGCAACAACCT
↪         [266..306[
        read 149:        CTGTGTTTAGTGCCTTTGTTCA-----ACCCCCTTGCAAC

ALIGNMENT 144
        contig 0:        AGTGCCTTTGTTCA-----ACCCCCTTGCAACAACCTTGAG
↪         [274..310[
        read 153:        AGTGCCTTTGTTCACATAGACCCCCTTGCAACAACC

ALIGNMENT 148
        contig 0:        TTCA-----ACCCCCTTGCAACAACCTTGAGAACCCCAGGGAATT
↪         [284..324[
        read 157:        ATAG-----ACCCCCTTGCAACAACCTTGAGAACCCCAGG

ALIGNMENT 152
        contig 0:        CA-----ACCCCCTTGCAACAACCTTGAGAACCCCAGGGAATTTG
↪         [286..326[
        read 161:        CA-----ACCCCCTTGCAACAACCTTGCGAACCCCAGGGA

ALIGNMENT 156
        contig 0:        TTGCAACAACCTTGAGAACCCCAGGGAATTTGTCA             [294..
↪329[
        read 165:        CCCCCTTGCAACAACCTTGAGAACCCCAGGGAATT
```

Gene Annotation

Annotations are represented as a tree that at least contains a root node where all annotations of children or grandchildren of. A typical annotation tree looks as follows:

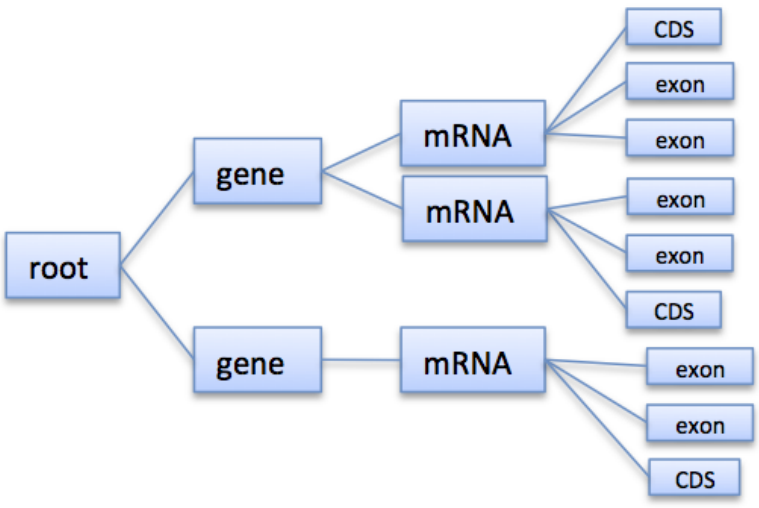


Fig. 4.7: ***Figure 4:*** Annotation tree example

The following figure shows which tables represent the annotation tree:

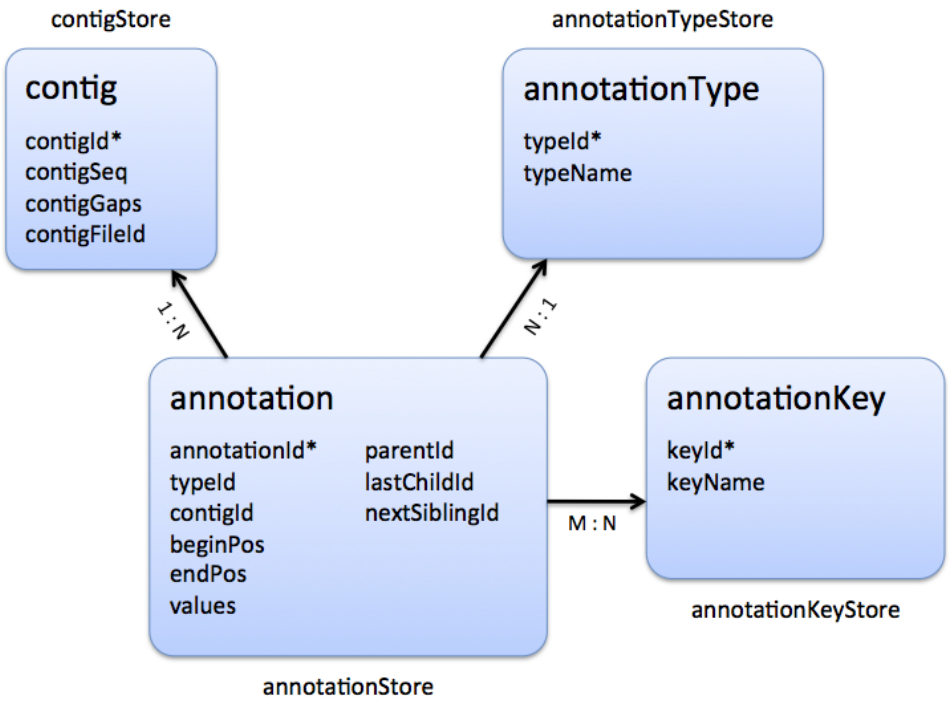


Fig. 4.8: ***Figure 5:*** Stores involved in gene annotation

Traversing the Annotation Tree

The annotation tree can be traversed and accessed with the AnnotationTree Iterator. A new iterator can be created with begin given a reference to the FragmentStore and the tag `AnnotationTree`:

```
Iterator<FragmentStore<>, AnnotationTree<> >::Type it;
it = begin(store, AnnotationTree<>());
```

It starts at the root node and can be moved to adjacent tree nodes with the functions goDown, goUp, and goRight. These functions return a boolean value that indicates whether the iterator could be moved. The functions isLeaf, isRoot, isLastChild return the same boolean without moving the iterator. With goRoot or goTo it can be moved to the root node or an arbitrary node given its annotationId. If the iterator should not be moved but a new iterator at an adjacent nodes is required, the functions nodeDown, nodeUp, nodeRight can be used.

The AnnotationTree iterator supports a preorder DFS traversal and therefore can also be used in typical begin-end loops with the functions goBegin (== goRoot), goEnd, goNext, atBegin, atEnd. During a preorder DFS, the descent into subtree can be skipped by goNextRight, or goNextUp which proceeds with the next sibling or returns to the parent node and proceeds with the next node in preorder DFS.

Accessing the Annotation Tree

To access or modify the node an iterator points at, the iterator returns the node's annotationId by the value function (== operator*). With the annotationId the corresponding entry in the annotationStore could be modified manually or by using convenience functions. The function getAnnotation returns a reference to the corresponding entry in the annotationStore. getName and setName can be used to retrieve or change the identifier of the annotation element. As some annotation file formats don't give every annotation a name, the function getUniqueName returns the name if non-empty or generates one using the type and id. The name of the parent node in the tree can be determined

with getParentName. The name of the annotation type, e.g. 'mRNA' or 'exon', can be determined and modified with getType and setType.

An annotation can not only reference a region of a contig but also contain additional information given as key-value pairs. The value of a key can be retrieved or set by getValueByKey and assignValueByKeq. The values of a node can be cleared with clearValues.

A new node can be created as first child, last child, or right sibling of the current node with createLeftChile, createRightChild, or createSibling. All three functions return an iterator to the newly created node.

The following tables summarizes the functions provided by the AnnotationTree iterator:

Function	Description
getAnnotation, value	Return annotation object/id of current node
[get/set]Name, [get/set]Type	Access name or type of current annotation object
clearValues, [get/set]ValueByKey	Access associated values
goBegin, goEnd, atBegin, atEnd	Go to or test for begin/end of DFS traversal
goNext, goNextRight, goNextUp	go next, skip subtree or siblings during DFS traversal
goRoot, goUp, goDown, goRight	Navigate through annotation tree
create[Left/Right]Child, createSibling	Create new annotation nodes
isRoot, isLeaf	Test for root/leaf node

File I/O

Reads and Contigs

To efficiently load reads, use the function loadReads which auto-detects the file format, supporting Fasta, Fastq, QSeq and Raw (see AutoSeqFormat), and uses memory mapping to efficiently load millions of reads, their names and quality values. If not only one but two file names are given, loadReads loads mate pairs or paired-end reads stored in two separate files. Both files are required to contain the same number or reads and reads stored at the same line in both files are interpreted as pairs. The function internally uses appendRead or appendMatePair and reads distributed over multiple files can be loaded with consecutive calls of loadReads.

Contigs can be loaded with the function loadContigs. The function loads all contigs given in a single file or multiple files given a single file name or a StringSet of file names. The function has an additional boolean parameter `loadSeqs` to load immediately load the contig sequence or if `false` load the sequence later with loadContig to save memory, given the corresponding `contigId`. If the contig is accessed by multiple instances/threads the functions lockContig and unlockContig can be used to ensure that the contig is loaded and release it after use. The function unlockAndFreeContig can be used to clear the contig sequence and save memory if the contig is not locked by any instance.

To write all contigs to an open output stream use writeContigs.

Multiple Read Alignments

A multiple read alignment can be loaded from an open BamFileIn with readRecords. Similarly, it can be written to an open BamFileOut with writeRecords.

As SAM supports a multiple read alignment (with padding operations in the CIGAR string) but does not enforce its use. That means that a typical SAM file represents a set of pairwise (not multiple) alignments. To convert all the pairwise alignments into a multiple alignments of all reads, read internally calls the function convertPairWiseToGlobalAlignment. A prior call to loadReads is not necessary (but possible) as SAM contains the read names, sequences and quality values. Contigs can be loaded at any time. If they are not loaded before reading a SAM file, empty sequences are created with the names referred in the SAM file. A subsequent call of loadContigs would load the sequences of these contigs, if they have the same identifier in the contig file.

Annotations

A annotation file can be read from an open GffFileIn or UcscFileIn with readRecords. Similarly, it can be written to an open GffFileOut with writeRecords.

The GffFileIn is also able to detect and read GTF files in addition to GFF files. As the kownGene.txt and knownIsoforms.txt files are two seperate files used by the UCSC Genome Browser, they must be read by two consecutive calls of readRecords (first knownGene.txt then knownIsoforms.txt). An annotation can be loaded without loading the corresponding contigs. In that case empty contigs are created in the contigStore with names given in the annonation. A subsequent call of loadContigs would load the sequences of these contigs, if they have the same identifier in the contig file.

Please note, that UCSC files cannot be written due to limitations of the file format.

Stores

The Fragment Store consists of the following tables:

Read Stores

Store	Description	Details
readStore	Reads	String mapping from `readId` to `matePairId`
readSeqStore	Read sequences	String mapping from `readId` to `readSeq`
matePairStore	Mate-pairs / pairs of reads	String mapping from `matePairId` to `<readId[2], libId>`
libraryStore	Mate-pair libraries	String mapping from `libId` to `<mean, std>`

Contig Stores

Store	Description	Details
contigStore	Contig sequences with gaps	String that maps from `contigId` to `<contigSeq, contigGaps, contigFileId>`
contigFileStore	Stores information how to load contigs on-demand	String that maps from `contigFileId` to `<fileName, firstContigId>`

Read Alignment Stores

Store	Description	Details
alignedReadStore	Alignments of reads against contigs	String that stores `<alignId, readId, contigId, pairMatchId, beginPos, endPos, gaps>`
alignedReadTagStore	Additional alignment tags (used in SAM)	String that maps from `alignId` to `alignTag`
alignQualityStore	Mapping quality of read alignments	String that maps from `alignId` to `<pairScore, score, errors>`

Annotation Stores

Store	Description	Details
annotation-Store	Annotations of contig regions	String that maps from `annoId` to `<contigId, typeId, beginPos, endPos, parentId, lastChildId, nextSiblingId, values>`

Name Stores

annotationNameStore	Annotation names	String that maps from `annoId` to `annoName`
readNameStore	Read identifiers (Fasta ID)	String that maps from `readId` to `readName`
contigNameStore	Contig identifiers (Fasta ID)	String that maps from `contigId` to `contigName`
matePairNameStore	Mate-pair identifiers	String that maps from `contigId` to `contigName`
libraryNameStore	Mate-pair library identifiers	String that maps from `libId` to `libName`

In this tutorial we will explain the GenomeAnnotation store and the FragmentStore. Both are very comprehensive and feature rich data structures that come with many advantages when dealing with annotation trees or read alignments. However, these are more advanced topics and rather aimed for more experienced SeqAn users.

ToC

Contents

Graphs

Learning Objective This tutorial shows how to use graphs in SeqAn and their functionality.

Difficulty Average

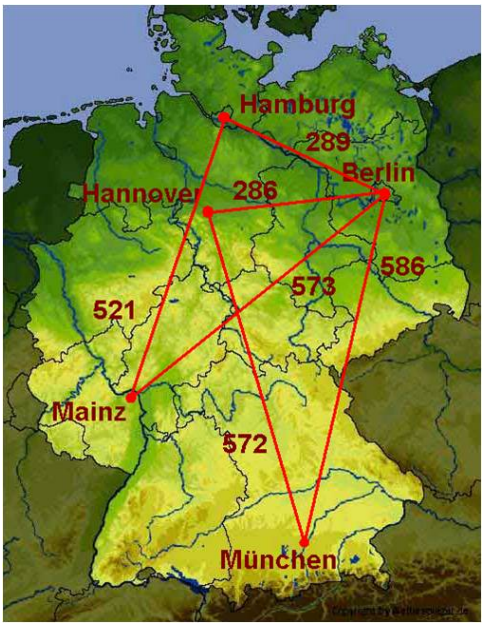
Duration 1 h

Prerequisites *Sequences*, *Alignment*, *Pairwise Sequence Alignment*

Overview

A graph in computer science is an ordered pair $G = (V, E)$ of a set of vertices V and a set of edges E. SeqAn provides different types of graphs and the most well-known graph algorithms as well as some specialized alignment graph algorithms. In this part of the tutorial, we demonstrate how to construct a graph in SeqAn and show the usage of some algorithms. Alignment graphs are described in the tutorial *Alignment*.

Let us follow a simple example. We have given the following network of five cities and in this network we want to compute the shortest path from Hannover to any other city.



In the section *Graph Basics*, we will create the network and write the graph to a *.dot* file. The section *Property Maps* assigns city names to the vertices and *Graph Iterators* demonstrates the usage of a vertex iterator.

After having worked through these sections you should be familiar with the general usage of graphs in SeqAn. You are then prepared to proceed with *Graph Algorithms*, where we will compute the shortest path by calling a single function.

Graph Basics

The general header file for all types of graphs is `<seqan/graph_types.h>`. It comprises the Graph class, its specializations, every function for basic graph operations and different iterators. Later, for computing the shortest path we will also need `<seqan/graph_algorithms.h>` which includes the implementations of most of SeqAn's graph algorithms.

```
#include <iostream>
#include <seqan/graph_types.h>
#include <seqan/graph_algorithms.h>
using namespace seqan;
```

We want to model the network of cities as an undirected graph and label the edges with distances. Before we start creating edges and vertices, we need some typedefs to specify the graph type.

SeqAn offers different specializations of the class Graph: Undirected Graph, DirectedGraph, Tree, Word Graph, Automaton, HmmGraph, and Alignment Graph. For our example, an undirected graph will be sufficient, so we define our own graph type `TGraph` with the specialization Undirected Graph of the class Graph. Luckily, this specialization has an optional cargo template argument, which attaches any kind of object to the edges of the graph. This enables us to store the distances between the cities, our edge labels, using the cargo type `TCargo` defined as `unsigned int`. Using the cargo argument, we have to provide a distance when adding an edge. And when we remove an edge we also remove the distance.

```
int main()
{
    typedef unsigned int TCargo;
```

```
typedef Graph<Undirected<TCargo> > TGraph;
typedef VertexDescriptor<TGraph>::Type TVertexDescriptor;
```

Each vertex and each edge in a graph is identified by a so-called descriptor. The type of the descriptors is returned by the metafunction VertexDescriptor. In our example, we define a type `TVertexDescriptor` by calling VertexDescriptor on our graph type. Analogously, there is the metafunction EdgeDescriptor for edge descriptors.

We can now create the graph `g` of our type `TGraph`.

```
TGraph g;
```

For our example, we add five vertices for the five cities, and six edges connecting the cities.

Vertices can be added to `g` by a call to the function addVertex. The function returns the descriptor of the created vertex. These descriptors are needed to add the edges afterwards.

```
TVertexDescriptor vertBerlin = addVertex(g);
TVertexDescriptor vertHamburg = addVertex(g);
TVertexDescriptor vertHannover = addVertex(g);
TVertexDescriptor vertMainz = addVertex(g);
TVertexDescriptor vertMuenchen = addVertex(g);
```

The function addEdge adds an edge to the graph. The arguments of this function are the graph to which the edge is added, the vertices that it connects, and the cargo (which is in our case the distance between the two cities).

```
addEdge(g, vertBerlin, vertHamburg, 289);
addEdge(g, vertBerlin, vertHannover, 286);
addEdge(g, vertBerlin, vertMainz, 573);
addEdge(g, vertBerlin, vertMuenchen, 586);
addEdge(g, vertHannover, vertMuenchen, 572);
addEdge(g, vertHamburg, vertMainz, 521);
```

Once we have created the graph we may want to have a look at it. SeqAn offers the possibility to write a graph to a dot file. With a tool like Graphviz you can then visualize the graph.

The only thing that we have to do is to call the function write on a file stream with the tag `DotDrawing()` and pass over our graph `g`.

```
std::ofstream dotFile("graph.dot");
writeRecords(dotFile, g, DotDrawing());
dotFile.close();
```

After executing this example, there should be a file `graph.dot` in your directory.

Alternatively, you can use the standard output to print the graph to the screen:

```
std::cout << g << '\n';
```

Assignment 1

Type Review

Objective Copy the code from above and adjust it such that a road trip from Berlin via Hamburg and Hannover to Munich is simulated.

Hints Use directed Edges

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/graph_types.h>
#include <seqan/graph_algorithms.h>
using namespace seqan;

int main()
{
    typedef unsigned int TCargo;
    typedef Graph<Directed<TCargo> > TGraph;
    typedef VertexDescriptor<TGraph>::Type TVertexDescriptor;

    TGraph g;

    TVertexDescriptor vertBerlin = addVertex(g);
    TVertexDescriptor vertHamburg = addVertex(g);
    TVertexDescriptor vertHannover = addVertex(g);
    TVertexDescriptor vertMuenchen = addVertex(g);

    addEdge(g, vertBerlin, vertHamburg, 289u);
    addEdge(g, vertHamburg, vertHannover, 289u);
    addEdge(g, vertHannover, vertMuenchen, 572u);

    std::ofstream dotFile("graph.dot");
    writeRecords(dotFile, g, DotDrawing());

    dotFile.close();

    std::cout << g << std::endl;

    return 0;
}
```

The output is the following:

```
Adjacency list:
0 -> 1,
1 -> 2,
2 -> 3,
3 ->
Edge list:
Source: 0,Target: 1 (Id: 0)
Source: 1,Target: 2 (Id: 1)
Source: 2,Target: 3 (Id: 2)
```

Assignment 2

Type Application

Objective Write a program which creates a directed graph with the following edges:

(1,0), (0,4), (2,1), (4,1), (5,1), (6,2), (3,2), (2,3), (7,3), (5,4), (6,5), (5,6), (7,6), (7,7)

Use the function addEdges instead of adding each edge separately. Output the graph to the screen.

Solution Click **more...** to see the solution.

We first have to include the corresponding header file for graphs. Instead of `<seqan/graph_types.h>`, we can also include `<seqan/graph_algorithms.h>` as it already includes `<seqan/graph_types.h>`.

```
#include <iostream>
#include <seqan/graph_algorithms.h>
using namespace seqan;
```

This time we define a DirectedGraph without cargo at the edges.

```
int main()
{
    typedef Graph<Directed<> > TGraph;
    typedef VertexDescriptor<TGraph>::Type TVertexDescriptor;
    typedef Size<TGraph>::Type TSize;
```

The function addEdges takes as parameters an array of vertex descriptors and the number of edges. The array of vertex descriptors is sorted in the way predecessor1, successor1, predecessor2, successor2, ...

```
    TSize numEdges = 14;
    TVertexDescriptor edges[] = {1,0, 0,4, 2,1, 4,1, 5,1, 6,2, 3,2, 2,3, 7,3, 5,4,
↪ 6,5, 5,6, 7,6, 7,7};
    TGraph g;
    addEdges(g, edges, numEdges);
    std::cout << g << std::endl;
```

The screen output of the graph consists of an adjacency list for the vertices and an edge list:

```
Adjacency list:
0 -> 4,
1 -> 0,
2 -> 3,1,
3 -> 2,
4 -> 1,
5 -> 6,4,1,
6 -> 5,2,
7 -> 7,6,3,
Edge list:
Source: 0,Target: 4 (Id: 1)
Source: 1,Target: 0 (Id: 0)
Source: 2,Target: 3 (Id: 7)
Source: 2,Target: 1 (Id: 2)
Source: 3,Target: 2 (Id: 6)
Source: 4,Target: 1 (Id: 3)
Source: 5,Target: 6 (Id: 11)
Source: 5,Target: 4 (Id: 9)
Source: 5,Target: 1 (Id: 4)
Source: 6,Target: 5 (Id: 10)
Source: 6,Target: 2 (Id: 5)
Source: 7,Target: 7 (Id: 13)
Source: 7,Target: 6 (Id: 12)
Source: 7,Target: 3 (Id: 8)
```

Assignment 3

Type Transfer

Objective Write a program which defines an HMM for DNA sequences:

- Define an **exon**, **splice**, and **intron** state.

- Sequences always start in the exon state. The probability to stay in an exon or intron state is **0.9**. There is exactly one switch from exon to intron. Between the switch from exon to intron state, the HMM generates exactly one letter in the splice state. The sequence ends in the intron state with a probability of **0.1**.
- Consider to use the type LogProb for the transition probabilities.
- Output the HMM to the screen.
- Use the follwing emission probabilities.

	A	C	G	T
exon state	0.25	0.25	0.25	0.25
splice state	0.05	0.0	0.95	0.0
intron state	0.4	0.1	0.1	0.4

Solution The program starts with the inclusion of `<seqan/graph_algorithms.h>`. In this example you could include `<seqan/graph_types.h>` instead of the algorithms header file. However, it is likely that if you define a graph, you will call a graph algorithm as well.

```
#include <iostream>
#include <seqan/graph_algorithms.h>

using namespace seqan;
```

Next, we define our types. The most interesting type here is `THmm`. It is a Graph with the specialization HmmGraph. The specialization takes itself three template arguments: the alphabet of the sequence that the HMM generates, the type of the transitions, and again a specialization. In our case, we define the transitions to be the logarithm of the probilities (LogProb) and hereby simplify multiplications to summations. For the specialization we explicitly use the `Default` tag. The default tag can always be omitted but it shows the possibility of further specialization.

```
int main()
{
    typedef LogProb<> TProbability;
    typedef Dna TAlphabet;
    typedef Size<TAlphabet>::Type TSize;
    typedef Graph<Hmm<TAlphabet, TProbability, Default> > THmm;
    typedef VertexDescriptor<THmm>::Type TVertexDescriptor;
```

After that, we define some variables, especially one of our type `THmm`.

```
    Dna dnaA = Dna('A');
    Dna dnaC = Dna('C');
    Dna dnaG = Dna('G');
    Dna dnaT = Dna('T');

    THmm hmm;
```

Now we can start with defining the states. States are represented by the vertices of the HMM-specialized graph.

The initial and terminating states of an HMM in SeqAn are always silent, i.e. they do not generate characters. That is why we have to define an extra begin state and tell the program that this is the initial state of the HMM. The latter is done by calling the function assignBeginState.

```
    TVertexDescriptor begState = addVertex(hmm);
    assignBeginState(hmm, begState);
```

For our three main states we also add a vertex to the HMM with addVertex. Additionally, we assign the emission probabilities for all possible characters of our alphabet using emissionProbability.

```
    TVertexDescriptor exonState = addVertex(hmm);
    emissionProbability(hmm, exonState, dnaA) = 0.25;
    emissionProbability(hmm, exonState, dnaC) = 0.25;
    emissionProbability(hmm, exonState, dnaG) = 0.25;
    emissionProbability(hmm, exonState, dnaT) = 0.25;

    TVertexDescriptor spliceState = addVertex(hmm);
    emissionProbability(hmm, spliceState, dnaA) = 0.05;
    emissionProbability(hmm, spliceState, dnaC) = 0.0;
    emissionProbability(hmm, spliceState, dnaG) = 0.95;
    emissionProbability(hmm, spliceState, dnaT) = 0.0;

    TVertexDescriptor intronState = addVertex(hmm);
    emissionProbability(hmm, intronState, dnaA) = 0.4;
    emissionProbability(hmm, intronState, dnaC) = 0.1;
    emissionProbability(hmm, intronState, dnaG) = 0.1;
    emissionProbability(hmm, intronState, dnaT) = 0.4;
```

Finally, we need to define the end state and call assignEndState.

```
    TVertexDescriptor endState = addVertex(hmm);
    assignEndState(hmm, endState);
```

For the HMM, only the transition probabilities are still missing. A transition is represented by an edge of our HMM graph type. The cargo on these edges correspond to the transition probabilities.

Since the sequences always start with an exon, we set the transition probability from the begin state to the exon state to 1.0 calling the already well-known function addEdge. And also the other transitions can be defined in the same way.

```
    addEdge(hmm, begState, exonState, 1.0);
    addEdge(hmm, exonState, exonState, 0.9);
    addEdge(hmm, exonState, spliceState, 0.1);
    addEdge(hmm, spliceState, intronState, 1.0);
    addEdge(hmm, intronState, intronState, 0.9);
    addEdge(hmm, intronState, endState, 0.1);
```

To check the HMM we can simply output it to the screen:

```
    std::cout << hmm << std::endl;
```

This should yield the following:

```
Alphabet:
{A,C,G,T}
States:
{0 (Silent),1,2,3,4 (Silent)}
Begin state: 0
End state: 4
Transition probabilities:
0 -> 1 (1)
1 -> 2 (0.1) ,1 (0.9)
2 -> 3 (1)
3 -> 4 (0.1) ,3 (0.9)
4 ->
Emission probabilities:
1: A (0.25) ,C (0.25) ,G (0.25) ,T (0.25)
```

```
2: A (0.05) ,C (0) ,G (0.95) ,T (0)
3: A (0.4) ,C (0.1) ,G (0.1) ,T (0.4)
```

Property Maps

So far, the vertices in our graph can only be distinguished by their vertex descriptor. We will now see how to associate the city names with the vertices.

SeqAn uses Property Maps to attach auxiliary information to the vertices and edges of a graph. The cargo parameter that we used above associated distances to the edges. In most scenarios you should use an external property map to attach information to a graph. Be aware that the word external is a hint that the information is stored independently of the graph and functions like removeVertex do not affect the property map. Property maps are simply Strings of a property type and are indexed via the already well-known vertex and edge descriptors.

Lets see how we can define a vertex property map for the city names. Our property type is a String of a city name, more explicitly a char string. The vertex property map should hold several names so we define a String of Strings. Now, we only have to create and resize this map so that it can hold information on all vertices.

```
    typedef String<char> TCityName;
    typedef String<TCityName> TProperties;
    TProperties cityNames;
    resizeVertexMap(cityNames, g);
```

Next, we can enter the city names for each vertex. Note that this is completely independent from our graph object `g`.

```
    assignProperty(cityNames, vertBerlin, "Berlin");
    assignProperty(cityNames, vertHamburg, "Hamburg");
    assignProperty(cityNames, vertMuenchen, "Munich");
    assignProperty(cityNames, vertMainz, "Mainz");
    assignProperty(cityNames, vertHannover, "Hannover");
```

If we now want to output all vertices including their associated information we can iterate through the graph and use the iterators value to access the information in the property map.

Graph Iterators

Let us have a quick look at iterators for graph types. SeqAn provides six different specializations for graph iterators: Vertex Iterator, Adjacency Iterator, Dfs Preorder Iterator, and Bfs Iterator for traversing vertices, and Edge Iterator and Out-edge Iterator for traversing edges. Except for the Vertex Iterator and the Edge Iterator who only depend on the graph, all other graph iterators depend additionally on a specified edge or vertex.

To output all vertices of our graph in an arbitrary order, we can define an iterator of the specialization Vertex Iterator and determine its type with the metafunction Iterator. The functions atEnd and goNext also work for graph iterators as for all other iterators in SeqAn.

The value of any type of vertex iterator is the vertex descriptor. To print out all city names we have to call the function getProperty on our property map `cityNames` with the corresponding vertex descriptor that is returned by the value function.

```
    typedef Iterator<TGraph, VertexIterator>::Type TVertexIterator;
    TVertexIterator itV(g);
    for (; !atEnd(itV); goNext(itV))
    {
        std::cout << value(itV) << ':' << getProperty(cityNames, value(itV)) <<␣
↪std::endl;
    }
```

The output of this piece of code should look as follows:

```
0:Berlin
1:Hamburg
2:Hannover
3:Mainz
4:Munich
```

Assignment 4

Type Application

Objective Add a vertex map to the program from assignment 2:

1. The map shall assign a lower-case letter to each of the seven vertices. Find a way to assign the properties to all vertices at once in a single function call (*without* using the function assignProperty for each vertex separately).
2. Show that the graph is not connected by iterating through the graph in depth-first-search ordering. Output the properties of the reached vertices.

Hint

- Use an array and the function assignVertexMap to assign all properties at once.
- Use the DFS Iterator for depth-first-search ordering.

Solution Our aim is to assign all properties at once to the vertices. Therefore, we create an array containing all the properties, the letters *'a'* through *'h'*.

The function assignVertexMap does not only resize the vertex map (as resizeVertexMap does) but also initializes it. If we specify the optional parameter `prop`, the values from the array `prop` are assigned to the items in the property map.

```
    String<char> nameMap;
    char names[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
    assignVertexMap(nameMap, g, names);
```

To iterate through the graph in depth-first-search ordering we have to define an Iterator with the specialization DfsPreorderIterator.

The vertex descriptor of the first vertex is `0` and we choose this vertex as a starting point for the depth-first-search through our graph `g` with the iterator `dfsIt`:

```
    TVertexDescriptor start = 0;
    typedef Iterator<TGraph, DfsPreorder>::Type TDfsIterator;
    TDfsIterator dfsIt(g, start);

    std::cout << "Iterate from '" << getProperty(nameMap, start) << "' in depth-
↪first-search ordering: ";
    while (!atEnd(dfsIt))
    {
        std::cout << getProperty(nameMap, getValue(dfsIt)) << ", ";
        goNext(dfsIt);
    }
    std::cout << std::endl;
```

For the chosen starting point, only two other vertices can be reached:

```
Iterate from 'a' in depth-first-search ordering: a, e, b,
```

ToC

Contents

Seeds

Learning Objective In this tutorial, you will learn about the seeds-related SeqAn class.

Difficulty Basic

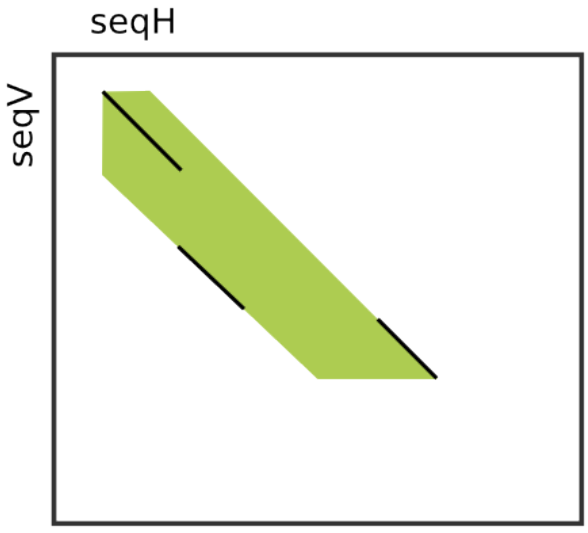
Duration 15 min

Prerequisites *Sequences*

Many efficient heuristics to find high scoring, but inexact, local alignments between two sequences start with small exact (or at least highly similar) segments, so called **seeds** and extend or combine them to get larger highly similar regions. Probably the most prominent tool of this kind is BLAST *[AGM+90]*, but there are many other examples like FASTA *[Pea90]* or LAGAN *[BDC+03]*. You will learn seed-and-extend and many applications including local and global chianing in *Seed Extension*.

SeqAn's header file for all data structures and functions related to two-dimensional seeds is `<seqan/seeds.h>`.

The Seed Class



The Seed class allows to store seeds. Seeds have a begin and end position in each sequence. Often, two or more close seeds are combined into a larger seed, possibly causing a shift in horizontal or vertical direction between the begin

position of the upper left seed and the end position of the lower right seed. For this reason, the Seed class also stores an upper and a lower diagonal to reflect the expansion between those shifted seeds.

The image to the right shows an example where three smaller seeds (black diagonals) were combined (or "chained locally") into one larger seed (green area).

The Simple Seed specialization only stores the begin and end positions of the seed (left-uppermost and right-lowermost corners of green surface) in both sequences and the upper and lower diagonal. The initial diagonals are not stored. The ChainedSeed specialization additionally stores these information. In most cases, the Simple Seed class is sufficient since the best alignment around the seeds has to be determined using a banded alignment algorithm of the seed infixes anyway.

You can get/set the begin and end position in the horizontal/vertical sequences using the functions beginPositionH, beginPositionV, setBeginPositionH, and setBeginPositionV. The band information can be queried and updated using lowerDiagonal, upperDiagonal, setLowerDiagonal, and setUpperDiagonal. Note, we use the capital letters 'H' and 'V' to indicate horizontal direction and vertical direction, respectively, while the **subject sequence** is always considered as the **horizontal sequence** and the **query** as the **vertical sequence** in the context of sequence alignments.

The following program gives an example of creating seeds as well as setting and reading properties.

```
    // Default-construct seed.
    Seed<Simple> seed1;
    std::cout << "seed1\n"
              << "beginPositionH == " << beginPositionH(seed1) << "\n"
              << "endPositionH == " << endPositionH(seed1) << "\n"
              << "beginPositionV == " << beginPositionV(seed1) << "\n"
              << "endPositionV == " << endPositionV(seed1) << "\n"
              << "lowerDiagonal == " << lowerDiagonal(seed1) << "\n"
              << "upperDiagonal == " << upperDiagonal(seed1) << "\n\n";

    // Construct seed with begin and end position in both sequences.
    Seed<Simple> seed2(3, 10, 7, 14);
    setUpperDiagonal(seed2, -7);
    setLowerDiagonal(seed2, -9);
    std::cout << "seed2\n"
              << "beginPositionH == " << beginPositionH(seed2) << "\n"
              << "endPositionH == " << endPositionH(seed2) << "\n"
              << "beginPositionV == " << beginPositionV(seed2) << "\n"
              << "endPositionV == " << endPositionV(seed2) << "\n"
              << "lowerDiagonal == " << lowerDiagonal(seed2) << "\n"
              << "upperDiagonal == " << upperDiagonal(seed2) << "\n\n";
```

The output to the console is as follows.

```
seed1
beginPositionH == 0
endPositionH == 0
beginPositionV == 0
endPositionV == 0
lowerDiagonal == 0
upperDiagonal == 0

seed2
beginPositionH == 3
endPositionH == 7
beginPositionV == 10
endPositionV == 14
lowerDiagonal == -9
upperDiagonal == -7
```

Assignment 1

Type Review

Objective Extend the program above such that `seed1` is updated from `seed2` and all members (begin positions, end positions, diagonals) are equal to the corresponding member of `seed` times two. For example, the lower diagonal of `seed2` should be `2 * lowerDiagonal(seed1)`.

Solution

```
#include <seqan/stream.h>
#include <seqan/seeds.h>
#include <seqan/sequence.h>

using namespace seqan;

int main()
{
    // Default-construct seed.
    Seed<Simple> seed1;

    // Construct seed with begin and end position in both sequences.
    Seed<Simple> seed2(3, 10, 7, 14);
    setUpperDiagonal(seed2, -7);
    setLowerDiagonal(seed2, -9);

    // Update seed1.
    setBeginPositionH(seed1, 2 * beginPositionH(seed2));
    setEndPositionH(seed1, 2 * endPositionH(seed2));
    setBeginPositionV(seed1, 2 * beginPositionV(seed2));
    setEndPositionV(seed1, 2 * endPositionV(seed2));
    setLowerDiagonal(seed1, 2 * lowerDiagonal(seed2));
    setUpperDiagonal(seed1, 2 * upperDiagonal(seed2));

    // Print resulting seed1.
    std::cout << "seed1\n"
              << "beginPositionH == " << beginPositionH(seed1) << "\n"
              << "endPositionH == " << endPositionH(seed1) << "\n"
              << "beginPositionV == " << beginPositionV(seed1) << "\n"
              << "endPositionV == " << endPositionV(seed1) << "\n"
              << "lowerDiagonal == " << lowerDiagonal(seed1) << "\n"
              << "upperDiagonal == " << upperDiagonal(seed1) << "\n\n";

    return 0;
}
```

ToC

Contents

Modifiers

Learning Objective In this tutorial you will learn how to modify the elements of a container without copying them using SeqAn modifiers. You will learn about the different specializations and how to work with them.

Difficulty Basic

Duration 20 min

Prerequisites *A First Example*, *Sequences*

Overview

Modifiers give a different view to other classes. They can be used to change the elements of a container without touching them. For example, someone gave you an algorithm that works on two arbitrary String objects, but you want to use it for the special pair of a string and its reverse (left-to-right mirror). The classical approach would be to make a copy of the one string, where all elements are mirrored from left to right and call the algorithm with both strings. With modifiers, e.g. a ModifiedString, you can create the reverse in $\mathcal{O}(1)$ extra memory without copying the original string. This can be handy if the original sequence is large.

Modifiers implement a certain concept (e.g. ContainerConcept, Iterator, ...) or class interface (String, ...) and thus can be used as such. The mirror modifier is already part of SeqAn and implements the class interface of String and can be used in every algorithm that works on strings.

The Modified String

The ModifiedString is a modifier that implements the String interface and thus can be used like a String. It has two template parameters. The first one specifies a sequence type (e.g. String, Segment, ...) and the second one specifies the modifiers behavior. That can be ModReverseString for mirroring a string left to right or ModViewModifiedString for applying a function to every single character (like 'C'->'G', 'A'->'T', ...).

ModReverse

We begin with the specialization ModReverseString from the example above. Now we have a given string:

```
#include <iostream>
#include <seqan/stream.h>
#include <seqan/modifier.h>

using namespace seqan;

int main()
{
    String<char> myString = "A man, a plan, a canal-Panama";
```

and want to get the reverse. So we need a ModifiedString specialized with `String<char>` and ModReverseString. We create the modifier and link it with `myString`:

```
    ModifiedString<String<char>, ModReverse> myModifier(myString);
```

The result is:

```
    std::cout << myString << std::endl;
    std::cout << myModifier << std::endl;
```

```
A man, a plan, a canal-Panama
amanaP-lanac a ,nalp a ,nam A
```

To verify that we didn't copy `myString`, we replace an infix of the original string and see that, as a side effect, the modified string has also changed:

```
    replace(myString, 9, 9, "master ");
    std::cout << myString << std::endl;
    std::cout << myModifier << std::endl;

    return 0;
}
```

```
A man, a master plan, a canal-Panama
amanaP-lanac a ,nalp retsam a ,nam A
```

ModView

Another specialization of the ModifiedString is the ModViewModifiedString modifier. Assume we need all characters of `myString` to be in upper case without copying `myString`. In SeqAn you first create a functor (a STL unary function) which converts a character to its upper-case character.

```
struct MyFunctor :
    public std::unary_function<char, char>
{
    inline char operator()(char x) const
    {
        if (('a' <= x) && (x <= 'z'))
            return x + ('A' - 'a');

        return x;
    }

};
```

and then create a ModifiedString specialized with `ModView<MyFunctor>`:

```
    String<char> myString = "A man, a plan, a canal-Panama";
    ModifiedString<String<char>, ModView<MyFunctor> > myModifier(myString);
```

The result is:

```
    std::cout << myString << std::endl;
    std::cout << myModifier << std::endl;
```

```
A man, a plan, a canal-Panama
A MAN, A PLAN, A CANAL-PANAMA
```

The upper-case functor and some other predefined functors are part of SeqAn (in `seqan/modifier/modifier_functors.h`) already. The following functors can be used as an argument of ModViewModifiedString:

`FunctorUpcase<TValue>` Converts each character of type `TValue` to its upper-case character

FunctorLowcase<TValue> Converts each character to type `TValue` to its lower-case character

FunctorComplement<Dna> Converts each nucleotide to its complementary nucleotide

FunctorComplement<Dna5> The same for the Dna5 alphabet

FunctorConvert<TInValue, TOutValue> Converts the type of each character from `TInValue` to `TOutValue`

So instead of defining your own functor we could have used a predefined one:

```
    ModifiedString< String<char>, ModView<FunctorUpcase<char> > >␣
↪myPredefinedModifier(myString);
```

Assignment 1

Type Review

Objective

In this assignment you will create a modifier using your own functor. Assume you have given two Dna sequences as strings as given in the code example below. Let's assume you know that in one of your Dna sequences a few 'C' nucleotides are converted into 'T' nucleotides, but you still want to compare the sequence. Extend the code example as follows:

1. Write a functor which converts all 'C' nucleotides to 'T' nucleotides.
2. Define a ModifiedString with the specialization ModViewModifiedString using this functor.
3. Now you can modify both sequences to compare them, treating all 'Cs' as 'Ts'. Print the results.

```
#include <iostream>
#include <seqan/stream.h>
#include <seqan/modifier.h>

using namespace seqan;

int main()
{
    typedef String<Dna> TSequence;

    TSequence seq1 = "CCCGGCATCATCC";
    TSequence seq2 = "CTTGGCATTATTC";

    std::cout << seq1 << std::endl;
    std::cout << seq2 << std::endl;
    std::cout << std::endl;

    return 0;
}
```

Solution

```
#include <iostream>
#include <seqan/stream.h>
#include <seqan/modifier.h>

using namespace seqan;

struct ConvertCT :
```

```
    public std::unary_function<Dna, Dna>
{
    inline Dna operator()(Dna x) const
    {
        if (x == 'C') return 'T';

        return x;
    }

};


int main()
{
    typedef String<Dna> TSequence;

    TSequence seq1 = "CCCGGCATCATCC";
    TSequence seq2 = "CTTGGCATTATTC";

    std::cout << seq1 << std::endl;
    std::cout << seq2 << std::endl;
    std::cout << std::endl;

    typedef ModifiedString<TSequence, ModView<ConvertCT> > TModCT;
    TModCT modCT1(seq1);
    TModCT modCT2(seq2);

    std::cout << modCT1 << std::endl;
    std::cout << modCT2 << std::endl;

    return 0;
}
```

```
CCCGGCATCATCC
CTTGGCATTATTC

TTTGGTATTATTT
TTTGGTATTATTT
```

For some commonly used modifiers you can use the following shortcuts:

Shortcut	Substitution
ModComplementDna	ModView<FunctorComplement<Dna> >
ModComplementDna5	ModView<FunctorComplement<Dna5> >
DnaStringComplement	ModifiedString<DnaString, ModComplementDna>
Dna5StringComplement	ModifiedString<Dna5String, ModComplementDna5>
DnaStringReverse	ModifiedString<DnaString, ModReverse>
Dna5StringReverse	ModifiedString<Dna5String, ModReverse>
DnaStringReverseComplement	ModifiedString<ModifiedString<DnaString, ModComplementDna>, ModReverse>
Dna5StringReverseComplement	ModifiedString<ModifiedString<Dna5String, ModComplementDna5>, ModReverse>

The Modified Iterator

We have seen how a ModifiedString can be used to modify strings without touching or copying original data. The same can be done with iterators. The ModifiedIterator implements the Iterator concept and thus can be used in every algorithm or data structure that expects an iterator. In fact, we have already used the ModifiedIterator unknowingly in the examples above, as in our cases the ModifiedString returns a corresponding ModifiedIterator via the Iterator meta-function. The main work is done in the ModifiedIterator, whereas the ModifiedString only overloads the begin and end. Normally, you are going to use the ModifiedString and maybe the result of its Iterator meta-function instead of a ModifiedIterator directly.

Nested Modifiers

As modifiers implement a certain concept and depend on classes of this concept, two modifiers can be chained to create a new modifier. We have seen how the ModifiedString specialized with ModReverseString and ModViewModifiedString can be used. Now we want to combine them to create a modifier for the reverse complement of a DnaString We begin with the original string:

```
    String<Dna> myString = "attacgg";
```

Then we define the modifier that complements a DnaString:

```
    typedef ModifiedString<String<Dna>, ModComplementDna>   TMyComplement;
```

This modifier now should be reversed from left to right:

```
    typedef ModifiedString<TMyComplement, ModReverse> TMyReverseComplement;
```

The original string can be given to the constructor.

```
    TMyReverseComplement myReverseComplement(myString);
```

The result is:

```
    std::cout << myString << '\n';
    std::cout << myReverseComplement << '\n';

    replace(myString, 1, 1, "cgt");

    std::cout << myString << '\n';
    std::cout << myReverseComplement << '\n';
```

```
ATTACGG
CCGTAAT
ACGTTTACGG
CCGTAAACGT
```

Using a predefined shortcut, the whole example could be reduced to:

```
    std::cout << DnaStringReverseComplement(myString) << std::endl;
```

ToC

Contents

Journaled String Tree

Learning Objective This tutorial introduces you to the new data structure Journaled String Tree. You will learn how to use this data structure and how to exploit it for an efficient analysis while searching multiple sequences simultaneously.

Difficulty Average

Duration 45 min

Prerequisites *Sequences*, *String Sets*

A typical task in bioinformatics is to find patterns in biological sequences e.g. transcription factors, or to examine different biological traits and the effects of modifications on such traits. With the current advances in sequencing technologies, sequences of whole populations have been made available. But the time for searching in all these sequences is proportional to the number of sequences searched. That's why it is important to find novel strategies to cope with the deluge of sequencing data. Since, many biological problems often involve the analysis of sequences of the same species, one effective strategy would be to exploit the similarities of such sequences.

For this special purpose we will introduce you to two data structures that are designed to improve these algorithmic tasks. The first one is the JournaledString and the second is the JournaledStringTree.

Journaled String

The JournaledString data structure behaves like a normal String in SeqAn, except that it is composed of two data structures.

1. The first data structure is a Holder which stores a sequence.
2. The second data structure stores modifications that are made to this particular sequence using a **journal** (see Journaling Filesystems for more information). This journal contains a list of deletions and insertions. The inserted characters are stored in an additional **insertion buffer**.

The advantage of this data structure lies in representing a String as a "patch" to another String. The journaled data structure can be modified without loosing the original context. We want to show you how to work with these data structures so you can build your own algorithms based on this.

First, we show you how to work with the Journaled String so you can learn the basic principles. To get access to the Journaled String implementation you have to include the `<seqan/sequence_journaled.h>` header file. Note that you will need the `<seqan/stream.h>` too in order to print the sequences.

```
#include <seqan/stream.h>
#include <seqan/sequence_journaled.h>

using namespace seqan;

int main()
{
```

In the next step we define the Journaled String type. A Journaled String is a specialization of the String class and is defined as `String<TValue, Journaled<THostSpec, TJournalSpec, TBufferSpec> >`. The specialization takes two parameters: (1) `TValue` defines the alphabet type used for the Journaled String and (2) `Journaled<>` selects the Journaled String specialization of the String class.

`Journaled<>` is further specialized with

- `THostSpec` selects the specialization of the underlying host sequence (`Alloc<>` for [dox:AllocString Alloc String),
- `TJournaleSpec` selects the used implementation to manage the journaled differences (here: `SortedArray`), and
- `TBufferSpec` selects the used specialization for the internally managed insertion buffer (here: `Alloc<>` as well).

In our scenario we use a `char` alphabet and [dox:AllocString Alloc String for the host string and the insertion buffer. Additionally, we use a `Sorted Array` as the model to manage the recorded differences.

We use the metafunction Host to get the type of the underlying host string used for the Journaled String.

```
typedef String<char, Journaled<Alloc<>, SortedArray, Alloc<> > > TJournaledString;
typedef Host<TJournaledString>::Type THost;
```

Now we can define the variables holding data structures. First, we construct our host sequence and after that we construct the Journaled String. Then, we set the host sequence using the function setHost. Afterwards, we examine the data structure in more detail and print the host sequence the constructed journaled sequence and the nodes of it.

```
THost hostStr = "thisisahostsequence";
TJournaledString journalStr;
setHost(journalStr, hostStr);

std::cout << "After creating the Journaled String:" << std::endl;
std::cout << "Host: " << host(journalStr) << std::endl;
std::cout << "Journal: " << journalStr << std::endl;
std::cout << "Nodes: " << journalStr._journalEntries << std::endl;
std::cout << std::endl;
```

Tip: The Journal

A node in the Journaled String represents either a part of the host sequence or a part of the insertion buffer. The type of a node is distinguished by the member variable **segmentSource** and can be of value `SOURCE_ORIGINAL` to refere to a part in the host or `SOURCE_PATCH` to refere to a part in the insertion buffer. A node further consists of three variables which specify the **virtual position**, the **physical position** and the **length** of this part. The **virtual position** gives the relative position of the Journaled String after all modifications before this position have been "virtually" applied. The **physical position** gives the absolute position where this part of the journal maps to either the host sequence or the insertion buffer.

This is followed by modifying our Journaled String. We insert the string `"modified"` at position `7` and delete the suffix `"sequence"` at position `19`. Note that position `19` refers to the string after the insertion of `"modified"` at position `7`. Again we print the host, the journaled sequence and the nodes that represent the modifications to see how our changes affect the host and the journaled sequence.

```
insert(journalStr, 7, "modified");
erase(journalStr, 19, 27);

std::cout << "After modifying the Journaled String:" << std::endl;
std::cout << "Host: " << host(journalStr) << std::endl;
```

```
    std::cout << "Journal: " << journalStr << std::endl;
    std::cout << "Nodes: " << journalStr._journalEntries << std::endl;
    std::cout << std::endl;
```

All of this is followed by calling flatten on our journeld string. This call applies all journaled changes to the host sequence. Again we print the sequences to see the effects.

```
    flatten(journalStr);
    std::cout << "After flatten the Journaled String:" << std::endl;
    std::cout << "Host: " << host(journalStr) << std::endl;
    std::cout << "Journal: " << journalStr << std::endl;
    std::cout << "Nodes: " << journalStr._journalEntries << std::endl;

    return 0;
}
```

Here is the output of our small program.

```
After creating the Journaled String:
Host: thisisahostsequence
Journal: thisisahostsequence
Nodes: JournalEntries({segmentSource=1, virtualPosition=0, physicalPosition=0,␣
↪physicalOriginPosition=0, length=19})

After modifying the Journaled String:
Host: thisisahostsequence
Journal: thisisamodifiedhost
Nodes: JournalEntries({segmentSource=1, virtualPosition=0, physicalPosition=0,␣
↪physicalOriginPosition=0, length=7}, {segmentSource=2, virtualPosition=7,␣
↪physicalPosition=0, physicalOriginPosition=0, length=8}, {segmentSource=1,␣
↪virtualPosition=15, physicalPosition=7, physicalOriginPosition=7, length=4})

After flatten the Journaled String:
Host: thisisamodifiedhost
Journal: thisisamodifiedhost
Nodes: JournalEntries({segmentSource=1, virtualPosition=0, physicalPosition=0,␣
↪physicalOriginPosition=0, length=19})
```

Important: Be careful when using the flatten function as it modifies the underlying host sequence. This might affect other journaled sequences that share the same host sequence. This becomes important especially when working with Journaled Sets where a whole set of sequences is journaled to the same reference.

Simultaneously Searching Multiple Sequences

Now, we come to a simple example to demonstrate the use of the Journaled String Tree (JST). As you could imagine, the JST internally uses a set of Journaled Strings to buffer the sequences, while requiring only a low memory footprint.

In this article, we are going to create a small JST, which we will use to search for a pattern using the Horspool algorithm.

Let's just start with the include headers. In order to make the JST implementation visible to our source code we need to include the header `include <seqan/journaled_string_tree.h>`.

```
#include <iostream>
#include <seqan/stream.h>
#include <seqan/journaled_string_tree.h>
```

In the next step, we are going to define the type of the JST.

```
using namespace seqan;

int main()
{
    typedef JournaledStringTree<DnaString> TJst;
    typedef Pattern<DnaString, Horspool>   TPattern;
    typedef Traverser<TJst>::Type          TTraverser;
```

The only information that is required is the type of the reference sequence used for the underlying sequence. We also defined the pattern type and a traverser type, which we will explain soon.

Now, we are ready to initialize the JST. To construct a JST, we need to know the reference sequence and how many sequences should be represented by the JST. In our case we assume 10 sequences. The JST supports insertion or deletion of delta events. A delta event is a tuple consisting of four parameters: The reference position, the value, the coverage and the delta type. The reference position determines the position within the reference sequence, where this event occurs. The value represents the actual modification applied to the sequences, that are determined by the coverage. The type of the value depends on the delta type.

Tip: The internal types, e.g. the types of the different delta events, of the JST can be overloaded with a second optional traits object. If no trait object is given DefaultJstConfig is taken as default. See the API documentation for more information.

The following listing creates a JST and inserts some delta events into the object:

```
    DnaString seq = "AGATCGAGCGAGCTAGCGACTCAG";
    TJst jst(seq, 10);

    insert(jst, 1, 3, std::vector<unsigned>{1, 3, 5, 6, 7}, DeltaTypeDel());
    insert(jst, 8, "CGTA", std::vector<unsigned>{1, 2}, DeltaTypeIns());
    insert(jst, 10, 'C', std::vector<unsigned>{4, 9}, DeltaTypeSnp());
    insert(jst, 15, 2, std::vector<unsigned>{0, 4, 7}, DeltaTypeDel());
    insert(jst, 20, 'A', std::vector<unsigned>{0, 9}, DeltaTypeSnp());
    insert(jst, 20, Pair<unsigned, DnaString>(1, "CTC"), std::vector<unsigned>{1, 2,␣
↪3, 7}, DeltaTypeSV());
```

After creating the JST, we can now prepare the search. To do so, we first define a needle that we want to search. Second, we need to instantiate a traverser object. A traverser represents the current state of the traversal over the JST. It is comparable to an iterator, but it is not lightweight, as it uses a state stack to implement the traversal over the JST. The traverser is initialized with two arguments: The instance of the JST and the context length, which is in our case the length of the needle.

Here is the listing:

```
    DnaString ndl = "AGCGT";
    TTraverser trav(jst, length(ndl));

    TPattern pat(ndl);
    JstExtension<TPattern> ext(pat);
```

In line 4 and 5 in the listing above we initialize the pattern with the needle and then create an `JstExtension` object.

This `JstExtension` is needed to extend the Pattern class of SeqAn with some auxiliary functions necessary for the JST based search. The only thing required, is that `pat` is fully initialized when passing it to `ext`.

The last preparation step we need before invoking the search algorithm is to create a functor that is called, whenever the search algorithm finds a match. In our scenario we simply want to print the sequences and the positions where the hit occurs. Therefor we create a simple `MatchPrinter` functor:

```
template <typename TTraverser>
struct MatchPrinter
{
    TTraverser & trav;

    MatchPrinter(TTraverser & _trav) : trav(_trav)
    {}

    void
    operator()()
    {
        auto pos = position(trav);
        for (auto p : pos)
        {
            std::cout  << "Seq: " << p.i1 << " Pos: " << p.i2 << std::endl;
        }
    }
};
```

This match printer, holds a reference to the actual traverser. So we can call the `position` function on the traverser, when the function-call-operator is invoked by the search algorithm.

Now we can invoke the search using the `find` interface:

```
    MatchPrinter<TTraverser> delegate(trav);
    find(trav, ext, delegate);

    return 0;
}
```

And finally the output:

```
Seq: 1 Pos: 7
Seq: 2 Pos: 10
```

The following list gives an overview of the available search algorithms:

Horspool Exact online search using HorspoolPattern as base pattern class.

ShiftAnd Exact online search using ShiftAndPattern as base pattern class.

ShiftOr Exact online search using ShiftOrPattern as base pattern class.

MyersUkkonen Approximate online search using MyersUkkonen as base pattern class.

Assignment 1

Type Review

Objective

Use the code from above and find all patterns of the needle `CCTCCA` with up to 2 errors.

Hints When searching with errors, the context size needs to be updated accordingly.

Solution Since we are trying to find the needle approximatively, we need to use the `Myers' bitvector` algorithm. Here is the entire solution:

```
#include <iostream>
#include <seqan/stream.h>
#include <seqan/journaled_string_tree.h>

template <typename TTraverser>
struct MatchPrinter
{
    TTraverser & trav;

    MatchPrinter(TTraverser & _trav) : trav(_trav)
    {}

    void
    operator()()
    {
        auto pos = position(trav);
        for (auto p : pos)
        {
            std::cout  << "Seq: " << p.i1 << " Pos: " << p.i2 << std::endl;
        }
    }
};

using namespace seqan;

int main()
{
    typedef JournaledStringTree<DnaString>   TJst;
    typedef Pattern<DnaString, MyersUkkonen> TPattern;
    typedef Traverser<TJst>::Type            TTraverser;

    DnaString seq = "AGATCGAGCGAGCTAGCGACTCAG";
    TJst jst(seq, 10);

    insert(jst, 1, 3, std::vector<unsigned>{1, 3, 5, 6, 7}, DeltaTypeDel());
    insert(jst, 8, "CGTA", std::vector<unsigned>{1, 2}, DeltaTypeIns());
    insert(jst, 10, 'C', std::vector<unsigned>{4, 9}, DeltaTypeSnp());
    insert(jst, 15, 2, std::vector<unsigned>{0, 4, 7}, DeltaTypeDel());
    insert(jst, 20, 'A', std::vector<unsigned>{0, 9}, DeltaTypeSnp());
    insert(jst, 20, Pair<unsigned, DnaString>(1, "CTC"), std::vector<unsigned>{1,␣
↪2, 3, 7}, DeltaTypeSV());

    DnaString ndl = "CCTCCA";
    TTraverser trav(jst, length(ndl) + 2);

    TPattern pat(ndl, -2);
    JstExtension<TPattern> ext(pat);

    MatchPrinter<TTraverser> delegate(trav);
    find(trav, ext, delegate);

    return 0;
}
```

And here is the output:

```
Seq: 7 Pos: 17
Seq: 7 Pos: 18
Seq: 4 Pos: 20
Seq: 1 Pos: 23
Seq: 2 Pos: 26
Seq: 3 Pos: 19
Seq: 1 Pos: 24
Seq: 2 Pos: 27
Seq: 3 Pos: 20
Seq: 1 Pos: 25
Seq: 2 Pos: 28
Seq: 3 Pos: 21
Seq: 7 Pos: 19
Seq: 1 Pos: 26
Seq: 2 Pos: 29
Seq: 3 Pos: 22
Seq: 7 Pos: 20
Seq: 5 Pos: 19
Seq: 6 Pos: 19
Seq: 8 Pos: 22
```

SeqAn has numerous data structures that are helpful for analyzing biological sequences. Those range from simple containers for strings that can be saved in different ways, to collection of strings or compressed strings.

The Journaled string tree, for example allows the user to traverse all sequence contexts, given a window of a certain size, that are present in a set of sequences. Similar sequences are hence only traversed once, and the coordinate bookkeeping is all within the data structure. This allows for example speedup of up to a 100x given sequences from the 1000 Genome project and compared to traversing the sequences one after another.

Another strong side of SeqAn are its generic string indices. You can think "suffix tree" but the implementations range from an enhanced suffix array to (bidirectional) FM-indices.

In this section you find tutorials addressing the most common of SeqAn's data structures.

The tutorials under *Sequences* will introduce you to alphabets, sequence containers, iterators and various kinds of string sets, among them also comrpessed, reference based representations. The tutorials under *Indices* will introduce you to the interfaces and implementations of SeqAn's string and q-gram indices and the corresponding interators.

The tutorials under *Alignment Representation (Gaps)* will introduce you to how SeqAn implements alignment objects (e.g. gaps in sequences). The tutorials under *Store* will introduce you to SeqAn's store data structures for NGS read mapping and annotation

The tutorials under *Graphs* will introduce you to SeqAn's graph type. Its simple and we provide in the algorithms section various standard graph algorithms for the datatype. The tutorials under *Seeds* will introduce you to seeds in SeqAn. Its what you need if you think 'seed-and-extend'.

The tutorials under *Modifiers* will introduce you SeqAn's modifier concept. If you want the reverse complement of a string, there is no need to explicitely allocate memory for it. Modifiers leave the sequence as it is but provide a different access to it.

The tutorials under *Journaled String Tree* will introduce you to a data structure which allows you to have data parallel access to a collection of similar strings. If you have an algorithm that scans collections of sequences one after another and left to right, this will be of interest for you, since it avoids a lot of redundant work if the sequences in the collection are similar.

Algorithms

Pattern Matching

ToC

Contents

Online Pattern Matching

Learning Objective In this tutorial you will learn how to use the SeqAn classes finder and pattern to search a known pattern in a string or StringSet.

Difficulty Average

Duration 40 min

Prerequisites *Sequences*, *Indices*

For all online search algorithms, the Finder is an iterator that scans over the `haystack`. The Pattern is a search algorithm dependent data structure preprocessed from the `needle`. The second template argument of the Pattern selects the search algorithm.

Exact Search

The following code snippet illustrates the usage of online search algorithms in SeqAn using the example of the Hoorspool algorithm *[Hor80]*. We begin by creating two strings of type `char` containing the `haystack` and the `needle`.

```
#include <iostream>
#include <seqan/find.h>

using namespace seqan;

int main()
{
    CharString haystack = "Simon, send more money!";
    CharString needle = "mo";
```

We then create Finder and Pattern objects of these strings and choose Horspool as the specialization in the second template argument of Pattern.

```
    Finder<CharString> finder(haystack);
    Pattern<CharString, Horspool> pattern(needle);
    while (find(finder, pattern))
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) <<
↪")\t" << infix(finder) << std::endl;
```

```
    return 0;
}
```

Program output:

```
[2,4)         mo
[12,14)         mo
[17,19)         mo
```

Currently the following exact online algorithms for searching a single sequence are implemented in SeqAn:

Simple Brute force algorithm

Horspool *[Hor80]*

Bfam Backward Factor Automaton Matching

BndmAlgo Backward Nondeterministic DAWG Matching

ShiftAnd Exact string matching using bit parallelism

ShiftOr Exact string matching using bit parallelism

... and for multiple sequences:

WuManber Extension of Horspool.

MultiBfam Multiple version of Bfam, uses an automaton of reversed needles.

SetHorspool Another extension of Horspool using a trie of reversed needles.

AhoCorasick *[AC75]*

MultipleShiftAnd Extension of ShiftAnd, should only be used if the sum of needle lengths doesn't exceed the machine word size.

Assignment 1

Type Review

Objective Use the given code example from below. Extend the code to search the given `haystack` simultaneously for "mo", "send" and "more". For every match output the begin and end position in the `haystack` and which `needle` has been found.

Hint Online search algorithms for multiple sequences simply expect needles of type `String<String<...> >`.

```
#include <iostream>
#include <seqan/find.h>

using namespace seqan;

int main()
{
    CharString haystack = "Simon, send more money!";
    String<CharString> needles;
    appendValue(needles, "mo");
    appendValue(needles, "send");
    appendValue(needles, "more");
```

```
    return 0;
}
```

You can use the specialization WuManber.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/find.h>

using namespace seqan;

int main()
{
    CharString haystack = "Simon, send more money!";
    String<CharString> needles;
    appendValue(needles, "mo");
    appendValue(needles, "send");
    appendValue(needles, "more");

    Finder<CharString> finder(haystack);
    Pattern<String<CharString>, WuManber> pattern(needles);
    while (find(finder, pattern))
    {
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) <
↪< ")\t";
        std::cout << position(pattern) << '\t' << infix(finder) << std::endl;
    }
    return 0;
}
```

We use a Pattern specialized with the WuManber algorithm for the search and initialize it with our `needles` string. For every match found by find we output the begin and end position and the match region in the `haystack` as well as the index of the found `needle` which is returned by `position(pattern)`.

```
[2,4)         0         mo
[7,11)         1         send
[12,14)         0         mo
[12,16)         2         more
[17,19)         0         mo
```

Approximate Search

The approximate search can be used to find segments in the `haystack` that are similar to a `needle` allowing errors, such as mismatches or indels. Note that if only mismatches are allowed, the difference of the end and begin position of a match is the length of the found `needle`. However, in the case of indels this difference may vary and is only a rough estimate for the length. Therefore, to find a begin position for a certain end position the findBegin interface should be used. The usage is similar to find and is shown in the next example. We want to find all semi-global alignments of a `needle` "more" with a SimpleScore of at least -2 using the scoring scheme (0,-2,-1) (match,mismatch,gap).

Again, we create `haystack` and `needle` strings first:

```
#include <iostream>
#include <seqan/find.h>

using namespace seqan;
```

```
int main()
{
    CharString haystack = "Simon, send more money!";
    CharString needle = "more";
```

We then create Finder and Pattern objects of these strings and choose DPSearch as the specialization in the second template argument of Pattern. DPSearch expects the scoring function as the first template argument which is SimpleScore in our example. The pattern is constructed using the `needle` as a template and our scoring object is initialized with the appropriate scores for match, mismatch and gap. As in the previous example, the main iteration uses find to iterate over all end positions with a minimum best score of -2. If such a semi-global alignment end position is found the begin position is searched via findBegin. Please note that we have to set the minimum score to the score of the match found (getScore) in order to find the begin of a best match. We then output all begin and end positions and the corresponding `haystack` segment for each match found.

```
    Finder<CharString> finder(haystack);
    Pattern<CharString, DPSearch<SimpleScore> > pattern(needle, SimpleScore(0, -2, -
↪1));
    while (find(finder, pattern, -2))
        while (findBegin(finder, pattern, getScore(pattern)))
            std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) <
↪< ")\t" << infix(finder) << std::endl;

    return 0;
}
```

Program output:

```
[2,4)         mo
[12,14)         mo
[12,15)         mor
[12,16)         more
[12,17)         more
[12,18)         more m
[17,19)         mo
[17,21)         mone
```

The following specializations are available:

Specialization DPSearch Dynamic programming algorithm for many kinds of scoring scheme

Specialization Myers *[Mye99]*, *[Ukk85]*

Specialization Pex *[BYN99]*

Specialization AbndmAlgo Approximate Backward Nondeterministic DAWG Matching, adaption of AbndmAlgo

Assignment 2

Type Application

Objective Use the example from above. Modify the code to search with the Myers algorithm for matches of `"more"` with an edit distance of at most 2.

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/find.h>
```

```
using namespace seqan;

int main()
{
    CharString haystack = "Simon, send more money!";
    CharString needle = "more";

    Finder<CharString> finder(haystack);
    Pattern<CharString, Myers<> > pattern(needle);
    while (find(finder, pattern, -2))
        while (findBegin(finder, pattern, getScore(pattern)))
            std::cout << '[' << beginPosition(finder) << ',' <<␣
↪endPosition(finder) << ")\t" << infix(finder) << std::endl;

    return 0;
}
```

We again set the `needle` to `"more"`. We then change the specialization tag of the Pattern to Myers with default arguments. As Myers algorithm is only applicable to edit distance searches it cannot be specialized or initialized with a scoring scheme. In SeqAn, edit distance corresponds to the scoring scheme (0,-1,-1) (match, mismatch, gap) and an edit distance of 2 corresponds to a minimum score of -2 given to the find function.

The program's output is as follows.

```
[2,4)         mo
[2,5)         mon
[2,6)         mon,
[12,14)         mo
[12,15)         mor
[12,16)         more
[12,17)         more
[12,18)         more m
[17,19)         mo
[17,20)         mon
[17,21)         mone
[17,22)         money
```

ToC

Contents

Indexed Pattern Matching

Learning Objective In this tutorial you will learn how to use the SeqAn classes finder and pattern to search a known pattern in a string or StringSet.

Difficulty Average

Duration 30 min

Prerequisites *Sequences*, *Indices*, *Online Pattern Matching*

Overview

The Finder is an object that stores all necessary information for searching for a pattern. We have learned how to use it in *Online Pattern Matching* tutorial.

The following line of code shows how the Finder is initialized with an index. In this example, we search for the pattern `ACGT`.

```
    String<Dna5> genome = "ACGTACGTACGTN";
    Index<String<Dna5>, IndexEsa<> > esaIndex(genome);
    Finder<Index<String<Dna5>, IndexEsa<> > > esaFinder(esaIndex);
```

Calling the function find invokes the localization of all occurrences of a given pattern. It works by modifying pointers of the `Finder` to tables of the index. For example, the Finder of `esaIndex` stores two pointers, pointing to the first and last suffix array entry that stores an occurrence of the pattern. The return value of the find function tells us whether or not a given pattern occurs in the text. Furthermore, if there are several instances of a pattern, consecutive calls of find will modify the Finder such that it points to the next occurrence after each call:

```
    find(esaFinder, "ACGT");  // first occurrence of "ACGT"
    find(esaFinder, "ACGT");  // second occurrence of "ACGT"
    find(esaFinder, "ACGT");  // third occurrence of "ACGT"
```

The above code is not very useful, since we do not know the locations of the first, second or third pattern occurrence. The function position will help here. position called on a finder returns the location of the `x`th pattern, where `x` can be the first, second, or any other occurrence of the pattern.

```
    find(esaFinder, "ACGT"); // first occurrence of "ACGT"
    position(esaFinder); // -> 0
    find(esaFinder, "ACGT"); // second occurrence of "ACGT"
    position(esaFinder); // -> 4
    find(esaFinder, "ACGT"); // third occurrence of "ACGT"
    position(esaFinder); // -> 8
```

Tip: Indices in SeqAn are built on demand. That means that the index tables are not build when the constructor is called, but when we search for a pattern for the first time.

Exact Search

For the index based search the Finder needs to be specialized with an Index of the `haystack` in the first template argument. The index itself requires two template arguments, the `haystack` type and a index specialization. In contrast, since the `needle` is not preprocessed the second template argument of the Pattern has to be omitted. The following source illustrates the usage of an index based search in SeqAn using the example of the IndexEsa index (an enhanced suffix array index). This is the default index specialization if no second template argument for the index is given. We begin to create an index object of our `haystack` `"tobeornottobe"` and a `needle` `"be"`.

```
int main()
{
    Index<CharString> index("tobeornottobe");
```

```
    CharString needle = "be";
    Finder<Index<CharString> > finder(index);
```

We proceed to create a Pattern of the needle and conduct the search in the usual way.

```
    Pattern<CharString> pattern(needle);
    while (find(finder, pattern))
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) <<
↪")\t" << infix(finder) << std::endl;
```

Instead of creating and using a pattern solely storing the `needle` we can pass the needle directly to find. Please note that an Index based Finder has to be reset with clear before conducting another search.

```
    clear(finder);
    while (find(finder, "be"))
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) <<
↪")\t" << infix(finder) << std::endl;

    return 0;
}
```

Program output:

```
[11,13)        be
[2,4)        be
[11,13)        be
[2,4)        be
```

All indices also support StringSet texts and can therefore be used to search multiple `haystacks` as the following example shows. We simply exchange the CharString of the haystack with a StringSet of CharString and append some strings to it.

```
int main()
{
    typedef StringSet<CharString> THaystacks;
    THaystacks haystacks;
    appendValue(haystacks, "tobeornottobe");
    appendValue(haystacks, "thebeeonthecomb");
    appendValue(haystacks, "beingjohnmalkovich");

    Index<THaystacks> index(haystacks);
    Finder<Index<THaystacks> > finder(haystacks);
```

The rest of the program remains unchanged.

```
    clear(finder);
    while (find(finder, "be"))
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) <<
↪")\t" << infix(finder) << std::endl;

    return 0;
}
```

```
[< 0 , 11 >,< 0 , 13 >)        be
[< 1 , 3 >,< 1 , 5 >)        be
[< 2 , 0 >,< 2 , 2 >)        be
[< 0 , 2 >,< 0 , 4 >)        be
```

The following index specializations support the Finder interface as described above.

Specialization IndexEsa Enhanced suffix array based index. Supports arbitrary needles.

Specialization IndexQGram Q-gram index. Needle mustn't exceed the size of the q-gram.

Specialization Open Adressing QGram Index Q-gram index with open addressing. Supports larger q-grams. Needle and q-gram must have the same size.

Besides the find interface there is another interface for indices using suffix tree iterators to search exact `needle` occurrences described in the tutorial *Indices*.

Assignment 1

Type Application

Objective Modify the example above to search with a Open Adressing QGram Index q-gram index for matches of "tobe" in "tobeornottobe".

Solution Click **more...** to see the solution.

```
#include <iostream>
#include <seqan/index.h>

using namespace seqan;

int main()
{
    typedef Index<CharString, IndexQGram<UngappedShape<4>, OpenAddressing> >␣
↪TIndex;
    TIndex index("tobeornottobe");
    Finder<TIndex> finder(index);

    while (find(finder, "tobe"))
        std::cout << '[' << beginPosition(finder) << ',' << endPosition(finder) <
↪< ")\t" << infix(finder) << std::endl;

    return 0;
}
```

We simply add a second template argument to the definition of the Index as described in the documentation of the Open Adressing QGram Index. As shape we can use an UngappedShape of length 4.

Program output:

```
[0,4)         tobe
[9,13)         tobe
```

Approximate Filtration

Currently there are no indices directly supporting an approximate search. But nevertheless, there are approximate search filters available that can be used to filter out regions of the `haystack` that do not contain an approximate match, see SwiftFinder and SwiftPattern. The regions found by these filters potentially contain a match and must be verified afterwards. beginPosition, endPosition and infix can be used to return the boundaries or sequence of such a potential match. For more details on using filters, see the article *Filtering Similar Sequences*.

Pattern matching is about searching a known string or StringSet (`needle`) in another string or StringSet (`haystack`). This tutorial will introduce you into the SeqAn classes Finder and Pattern. It will demonstrate how to use the spezializations of the class finder to perform either an online search or an index based seach. And you will learn how to specify the search algorithm, which can be either exact or approximate.

Overview

In the case of approximate searching errors are allowed, which are either only mismatches or also indels. Additionally there are filtration algorithms which return potential matches, i.e. `haystack` segments possibly containing a pattern match. All searching is done by calling the function find, which takes at least two arguments:

1. A Finder that stores all necessary information about the `haystack` and the last found position of the `needle` within the haystack.
2. A Pattern that stores all information about the `needle`. Some variants of find support further arguments. The Finder and Pattern classes expect the underlying `haystack` and `needle` types as first template arguments. In addition, a second template argument specifies the search algorithm.

Each call of find finds only one match (or potential match) of the `needle` within the haystack. The Finder can be asked for the begin and end position of the last found match. The Pattern can be asked for the number of the found sequence if the `needle` is a StringSet. Subsequent calls of find can be used to find more occurrences of the `needle`, until no more occurrences can be found and find returns `false`.

In general, search algorithms can be divided into algorithms that preprocess the `needle` (online search) or preprocess the `haystack` (index search).

DP Alignment

ToC

Contents

Pairwise Sequence Alignment

Learning Objective You will learn how to compute global and local alignments, how you can use different scoring schemes, and how you can customize the alignments to fulfill your needs.

Difficulty Average

Duration 1h

Prerequisites *A First Example*, *Sequences*, *Scoring Schemes*, *Graphs*

Alignments are one of the most basic and important ways to measure similarity between two or more sequences. In general, a pairwise sequence alignment is an optimization problem which determines the best transcript of how one sequence was derived from the other. In order to give an optimal solution to this problem, all possible alignments between two sequences are computed using a **Dynamic Programming** approach. Scoring schemes allow the comparison of the alignments such that the one with the best score can be picked. Despite of the common strategy to compute an alignment, there are different variations of the standard DP algorithm laid out for special purposes.

We will first introduce you to the global alignments. Subsequent, you will learn how to compute local alignments. Finally, we will demonstrate how you can reduce the search space using a band.

Global Alignments

In this section, we want to compute a global alignment using the Needleman-Wunsch algorithm. We will use the Levenshtein distance as our scoring scheme.

A program always starts with including the headers that contain the components (data structures and algorithms) we want to use. To gain access to the alignment algorithms we need to include the `<seqan/align.h>` header file. We tell the program that it has to use the `seqan` namespace and write the `main` function with an empty body.

A good programming practice is to define all types that shall be used by the function at the beginning of the function body. In our case, we define a `TSequence` type for our input sequences and an Align object (`TAlign`) type to store the alignment. For more information on the Align datastructure, please read the tutorial *Alignment Representation (Gaps)*.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;                 // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;     // align type
```

After we defined the types, we can define the variables and objects. First, we create two input sequences `seq1 = "CDFGHC"` and `seq2 = "CDEFGAHC"`. We then define an 'align' object where we want to put the sequences into, we resize it to manage two Gaps objects, and then assign the sequences to it.

```
    TSequence seq1 = "CDFGHC";
    TSequence seq2 = "CDEFGAHC";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);
```

Now, we can compute our first alignment. To do so, we simply call the function globalAlignment and give as input parameters the `align` object and the scoring scheme representing the Levenshtein distance. The globalAlignment function returns the score of the best alignment, which we store in the `score` variable. Afterwards, we print the computed score and the corresponding alignment.

```
    int score = globalAlignment(align, Score<int, Simple>(0, -1, -1));
    std::cout << "Score: " << score << std::endl;
    std::cout << align << std::endl;
```

```
    return 0;
}
```

The output is as follows:

```
Score: -2
      0     .
        CD-FG-HC
        || || ||
        CDEFGAHC
```

Assignment 1

Type Review

Objective

> Compute two global alignments between the DNA sequences `"AAATGACGGATTG"`. `"AGTCGGATCTACTG"` using the Gotoh algorithm *[Got82]*, implementing the Affine Gap model, with the following scoring parameters: `match = 4`, `mismatch = -2`, `gapOpen = -4` and `gapExtend = -2`. Store the alignments in two Align objects and print them together with the scores.

Hints The Gotoh algorithm uses the Affine Gap function. In SeqAn you can switch between Linear, Affine and Dynamic gap functions by customizing your scoring scheme with one of the three tags `LinearGaps()`, `AffineGaps()` or `DynamicGaps()` and relative penalty values `gapOpen` and `gapExtend`. When a single gap value is provided the Linear Gap model is selected as default while the Affine Gap model is chosen as standard when two different gap costs are set. If the Dynamic Gap model *[UPA+14]* is required the relative tag must be supplied. Have a look on the *Scoring Schemes* section if you are not sure about the correct ordering.

Solution First we have to define the body of our program. This includes the definition of the library headers that we want to use. In this case it is the *iostream* from the STL and the *<seqan/align.h>* header file defining all algorithms and data structures we want to use. After we added the namespace and opened the *main* body we define our types we want to use in this function. We use an String with the Dna alphabet, since we know that we work with DNA sequences. The second type is our Align object storing the alignment later on.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<Dna> TSequence;                 // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;    // align type
```

In the next step we initialize our objects. This includes the both input sequences `seq1` and `seq2` and `align`. We resize the underlying set of `align` that manages the separate Gaps data structures. Finally, we assign the input sequences as sources to the corresponding row of `align`.

```
    TSequence seq1 = "AAATGACGGATTG";
    TSequence seq2 = "AGTCGGATCTACTG";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);
```

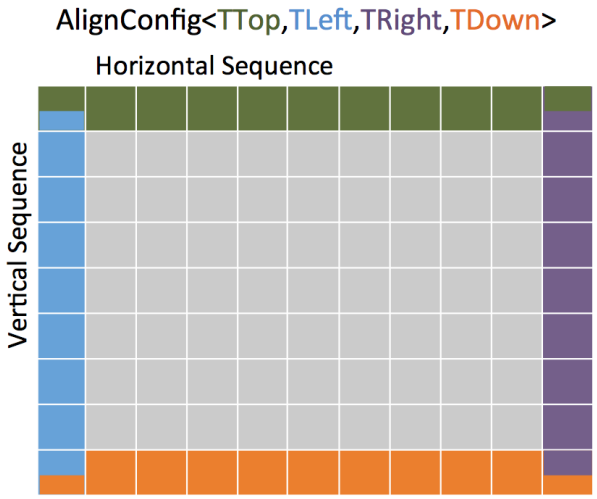
Now we compute the alignment using a scoring scheme with affine gap costs. The first parameter corresponds to the `match` value, the second to the `mismatch` value, the third to the `gap extend` value and the last one to the `gap open` value. We store the computed score of the best alignment in the equally named variable `score`. In the end we print the score and the alignment using print methods provided by the `iostream` module of the STL.

```
    int score = globalAlignment(align, Score<int, Simple>(4, -2, -2, -4),␣
↪AffineGaps());
    std::cout << "Score: " << score << std::endl;
    std::cout << align << std::endl;

    return 0;
}
```

Congratulation! You have computed an alignment using affine gap costs. Here the result of the program:

```
Score: 16
      0     .     :     .
        AAATGACGGAT----TG
        |   | |||||    ||
        A---GTCGGATCTACTG
```

Overlap Alignments



In contrast to the global alignment, an overlap alignment does not penalize gaps at the beginning and at the end of the sequences. This is referred to as **free end-gaps**. It basically means that overlap alignments can be shifted such that the end of the one sequence matches the beginning of the other sequence, while the respective other ends are gapped.

We use the AlignConfig object to tell the algorithm which gaps are free. The AlignConfig object takes four explicitly defined bool arguments. The first argument stands for `initial gaps` in the vertical sequence of the alignment matrix (first row) and the second argument stands for `initial gaps` in the horizontal sequence (first column). The third parameter stands for `end` gaps in the horizontal sequence (last column) and the fourth parameter stands for `end gaps` in the vertical sequence (last row). Per default the arguments of AlignConfig are set to `false` indicating a standard global alignment as you have seen above. In an overlap alignment all arguments are set to `true`. This means the first row and first column are initialized with zeros and the maximal score is searched in the last column and in the last row.

Just let us compute an overlap alignment to see how it works. We will also make use of the Alignment Graph to store

the alignment this time. We start again with including the necessary headers and defining all types that we need. We define the `TStringSet` type to store our input sequences in a StringSet and we define the `TDepStringSet` which is an DependentStringSet used internally by the AlignmentGraph.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;                                 // sequence type
    typedef StringSet<TSequence> TStringSet;                        // container for
↪strings
    typedef StringSet<TSequence, Dependent<> > TDepStringSet;    // dependent string
↪set
    typedef Graph<Alignment<TDepStringSet> > TAlignGraph;        // alignment graph
```

Before we can initialize the AlignmentGraph we append the input sequences to the StringSet `strings`. Then we simply pass `strings` as an argument to the constructor of the AlignmentGraph `alignG`.

```
    TSequence seq1 = "blablubalu";
    TSequence seq2 = "abba";

    TStringSet sequences;
    appendValue(sequences, seq1);
    appendValue(sequences, seq2);

    TAlignGraph alignG(sequences);
```

Now we are ready to compute the alignment. This time we change two things when calling the `globalAlignment` function. First, we use an AlignmentGraph to store the computed alignment and second we use the AlignConfig object to compute the overlap alignment. The gap model tag can be provided as last argument.

```
    int score = globalAlignment(alignG, Score<int, Simple>(1, -1, -1), AlignConfig
↪<true, true, true, true>(), LinearGaps());
    std::cout << "Score: " << score << std::endl;
    std::cout << alignG << std::endl;

    return 0;
}
```

The output is as follows.

```
Score: 2
Alignment matrix:
      0     .    :
        blablubalu
          ||  ||
        --ab--ba--
```

Assignment 2

Type Review

Objective Compute a semi-global alignment between the sequences `AAATGACGGATTG` and `TGGGA` using the costs 1, -1, -1 for match, mismatch and gap, respectively. Use an AlignmentGraph to store the alignment. Print the

score and the resulting alignment to the standard output.

Hint A semi-global alignment is a special form of an overlap alignment often used when aligning short sequences against a long sequence. Here we only allow free end-gaps at the beginning and the end of the shorter sequence.

Solution First we have to define the body of our program. This includes the definition of the library headers that we want to use. In this case we include the `iostream` header from the STL and the `<seqan/align.h>` header, which defines all algorithms and data structures we want to use. After we added the namespace and opened the `main` function body we define our types we want to use in this function. We use an String with the Dna alphabet, since we know that we work with DNA sequences. We use an additional StringSet to store the input sequences. In this scenario we use an AlignmentGraph to store the alignment. Remember, that the AlignmentGraph uses an DependentStringSet to map the vertices to the correct input sequences.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<Dna> TSequence;                                  // sequence type
    typedef StringSet<TSequence> TStringSet;                        // container for␣
↪strings
    typedef StringSet<TSequence, Dependent<> > TDepStringSet;       // dependent␣
↪string set
    typedef Graph<Alignment<TDepStringSet> > TAlignGraph;           // alignment graph
```

In the next step we initialize our input StringSet `strings` and pass it as argument to the constructor of the AlignmentGraph `alignG`.

```
    TSequence seq1 = "AAATGACGGATTG";
    TSequence seq2 = "TGGGA";

    TStringSet sequences;
    appendValue(sequences, seq1);
    appendValue(sequences, seq2);

    TAlignGraph alignG(sequences);
```

Now we compute the alignment using the Levenshtein distance and a AlignConfig object to set the correct free end-gaps. In this example we put the shorter sequence on the vertical axis of our alignment matrix. Hence, we have to use free end-gaps in the first and last row, which corresponds to the first and the last parameter in the AlignConfig object. If you add the shorter sequence at first to `strings`, then you simply have to flip the `bool` values of the AlignConfig object.

```
    int score = globalAlignment(alignG, Score<int, Simple>(1, -1, -1), AlignConfig
↪<true, false, false, true>());
    std::cout << "Score: " << score << std::endl;
    std::cout << alignG << std::endl;

    return 0;
}
```

Here the result of the program.

```
Score: 3
Alignment matrix:
      0     .     :
```

```
        AAATGACGGATTG
           ||  |||
        ---TG--GGA---
```

Specialized Alignments

SeqAn offers specialized algorithms that can be selected using a tag. Note that often these specializations are restricted in some manner. The following list shows different alignment tags for specialized alignment algorithms and the restrictions of the algorithms.

Hirschberg The Hirschberg algorithm computes an alignment between two sequences in linear space. The algorithm can only be used with an Align object (or Gaps). It uses only linear gap costs and does no overlap alignments.

MyersBitVector The MyersBitVector is a fast alignment specialization using bit parallelism. It only works with the Levenshtein distance and outputs no alignments.

MyersHirschberg The MyersHirschberg is an combination of the rapid MyersBitVector and the space efficient Hirschberg algorithm, which additionally enables the computation of an alignment. It only works with the Levenshtein distance and for Align objects.

Tip: In SeqAn you can omit the computation of the traceback to get only the score by using the function globalAlignmentScore. This way you can use the alignment algorithms for verification purposes, etc.

In the following example, we want to compute a global alignment of two sequences using the Hirschberg algorithm. We are setting the `match` score to `1`, and `mismatch` as well as `gap` penalty to `-1`. We print the alignment and the score.

First the necessary includes and typedefs:

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;                 // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;     // align type

    TSequence seq1 = "GARFIELDTHECAT";
    TSequence seq2 = "GARFIELDTHEBIGCAT";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);
```

In addition to the previous examined examples we tell the globalAlignment function to use the desired Hirschberg algorithm by explicitly passing the tag `Hirschberg` as last parameter. The resulting alignment and score are then printed.

```
    int score = globalAlignment(align, Score<int, Simple>(1, -1, -1), Hirschberg());
    std::cout << "Score: " << score << std::endl;
    std::cout << align << std::endl;
```

```
    return 0;
}
```

The output is as follows.

```
Score: 11
      0     .    :    .
        GARFIELDTHE---CAT
        ||||||||||   |||
        GARFIELDTHEBIGCAT
```

Assignment 3

Type Application

Objective Write a program that computes a global alignment between the Rna sequences `AAGUGACUUAUUG` and `AGUCGGAUCUACUG` using the Myers-Hirschberg variant. You should use the Align data structure to store the alignment. Print the score and the alignment. Additionally, output for each row of the Align object the view positions of the gaps.

Hint You can use an iterator to iterate over a row. Use the metafunction Row to get the type of the row used by the Align object. Use the function isGap to check whether the current value of the iterator is a gap or not. The gaps are already in the view space.

Solution As usual, first the necessary includes and typedefs. Our sequence type is `String<Rna>`. `TAlign` and `TRow` are defined as in the previous example program. The type `Iterator<TRow>::Type` will be used to iterate over the rows of the alignment.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<Rna> TSequence;
    typedef Align<TSequence, ArrayGaps> TAlign;
    typedef Row<TAlign>::Type TRow;
    typedef Iterator<TRow>::Type TRowIterator;
```

In the next step we initialize our Align object `align` with the corresponding source files.

```
    TSequence seq1 = "AAGUGACUUAUUG";
    TSequence seq2 = "AGUCGGAUCUACUG";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);
```

Now we compute the alignment using Myers-Hirschberg algorithm by specifying the correct tag at the end of the function.

```
    int score = globalAlignment(align, MyersHirschberg());
    std::cout << "Score: " << score << std::endl;
    std::cout << align << std::endl;
```

Finally, we iterate over both gap structures and print the view positions of the gaps within the sequences.

```
    unsigned aliLength = _max(length(row(align, 0)), length(row(align, 1)));
    for (unsigned i = 0; i < length(rows(align)); ++i)
    {
        TRowIterator it = iter(row(align, i), 0);
        TRowIterator itEnd = iter(row(align, i), aliLength);
        unsigned pos = 0;
        std::cout << "Row " << i << " contains gaps at positions: ";
        std::cout << std::endl;
        while (it != itEnd)
        {
            if (isGap(it))
                std::cout << pos << std::endl;
            ++it;
            ++pos;
        }
    }

    return 0;
}
```

The output of the program is as follows.

```
Score: -6
      0     .    :    .
        AAGU--GA-CUUAUUG
        | ||  || || | ||
        A-GUCGGAUCU-ACUG



Row 0 contains gaps at positions:
4
5
8
Row 1 contains gaps at positions:
1
11
```

Local Alignments

Now let's look at local pairwise alignments.

SeqAn offers the classical Smith-Waterman algorithm that computes the best local alignment with respect to a given scoring scheme, and the Waterman-Eggert algorithm, which computes not only the best but also suboptimal local alignments.

We are going to demonstrate the usage of both in the following example where first the best local alignment of two character strings and then all local alignments of two DNA sequences with a score greater than or equal to 4 are computed.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;
```

```
int main()
{
```

Let's start with initializing the Align object to contain the two sequences.

```
    Align<String<char> > ali;
    resize(rows(ali), 2);
    assignSource(row(ali, 0), "aphilologicaltheorem");
    assignSource(row(ali, 1), "bizarreamphibology");
```

Now the best alignment given the scoring parameters is computed using the Dynamic Gap model by the function localAlignment. The returned score value is printed directly, and the alignment itself in the next line. The functions clippedBeginPosition and clippedEndPosition can be used to retrieve the begin and end position of the matching subsequences within the original sequences.

```
    std::cout << "Score = " << localAlignment(ali, Score<int>(3, -3, -2, -2),␣
↪DynamicGaps()) << std::endl;
    std::cout << ali;
    std::cout << "Aligns Seq1[" << clippedBeginPosition(row(ali, 0)) << ":" <<␣
↪(clippedEndPosition(row(ali, 0)) - 1) << "]";
    std::cout << " and Seq2[" << clippedBeginPosition(row(ali, 1)) << ":" << ␣
↪(clippedEndPosition(row(ali, 1)) - 1) << "]" << std::endl << std::endl;
```

Next, several local alignments of the two given DNA sequences are going to be computed. First, the Align object is created.

```
    Align<String<Dna> > ali2;
    resize(rows(ali2), 2);
    assignSource(row(ali2, 0), "ataagcgtctcg");
    assignSource(row(ali2, 1), "tcatagagttgc");
```

A LocalAlignmentEnumerator object needs to be initialized on the Align object. In addition to the Align object and the scoring scheme, we now also pass the `finder` and a minimal score value, 4 in this case, to the localAlignment function. The `WatermanEggert` tag specifies the desired Waterman-Eggert algorithm. While the score of the local alignment satisfies the minimal score cutoff, the alignments are printed with their scores and the subsequence begin and end positions.

```
    Score<int> scoring(2, -1, -2, 0);
    LocalAlignmentEnumerator<Score<int>, Unbanded> enumerator(scoring, 5);
    while (nextLocalAlignment(ali2, enumerator))
    {
        std::cout << "Score = " << getScore(enumerator) << std::endl;
        std::cout << ali2;
        std::cout << "Aligns Seq1[" << clippedBeginPosition(row(ali2, 0)) << ":" <<␣
↪(clippedEndPosition(row(ali2, 0)) - 1) << "]";
        std::cout << " and Seq2[" << clippedBeginPosition(row(ali2, 1)) << ":" << ␣
↪(clippedEndPosition(row(ali2, 1)) - 1) << "]" << std::endl << std::endl;
    }
    return 0;
}
```

Here is the output of our example program. The first part outputs one alignment. The second part outputs two alignments:

```
Score = 19
      0     .    :
        a-philolog
```

```
        | ||| ||||
        amphibolog


Aligns Seq1[0:9] and Seq2[7:16]

Score = 9
      0     .
        ATAAGCGT
        ||| | ||
        ATA-GAGT


Aligns Seq1[0:7] and Seq2[2:9]

Score = 5
      0     .
        TC-TCG
        || | |
        TCATAG


Aligns Seq1[7:12] and Seq2[0:5]
```

Assignment 4

Type Review

Objective Write a program which computes the 3 best local alignments of the two AminoAcid sequences "PNCFDAKQRTASRPL" and "CFDKQKNNRTATRDTA" using the following scoring parameters: `match = 3`, `mismatch = -2`, `gap open = -5`, `gap extension = -1`.

Hint Use an extra variable to enumerate the k best alignments.

Solution The usual includes.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
```

The initialization of the Align object.

```
    Align<String<AminoAcid> > ali;
    resize(rows(ali), 2);
    assignSource(row(ali, 0), "PNCFDAKQRTASRPL");
    assignSource(row(ali, 1), "CFDKQKNNRTATRDTA");
```

Computing the three best alignments with the desired scoring parameters:

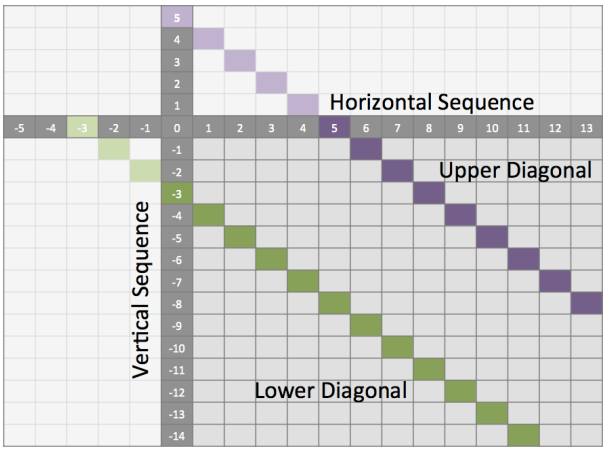
```
    Score<int> sc(3, -2, -1, -5);
    unsigned count = 0;
    LocalAlignmentEnumerator<Score<int>, Unbanded> enumerator(sc);
    while (nextLocalAlignment(ali, enumerator) && count < 3)
```

```
    {
        std::cout << "Score = " << getScore(enumerator) << std::endl;
        std::cout << ali;
        ++count;
    }
    return 0;
}
```

The resulting output is as follows.

```
Score = 21
      0     .    :
        CFDAKQ---RTASR
        ||| ||   ||| |
        CFD-KQKNNRTATR


Score = 8
      0     .
        AKQR-TA
        |  | ||
        AT-RDTA


Score = 5
      0
        D-A
        | |
        DTA
```

Banded Alignments



Often it is quite useful to reduce the search space in which the optimal alignment can be found, e.g., if the sequences are very similar, or if only a certain number of errors is allowed. To do so you can define a band, whose intersection with the alignment matrix defines the search space. To define a band we have to pass two additional parameters to the alignment function. The first one specifies the position where the lower diagonal of the band crosses the vertical axis. The second one specifies the position where the upper diagonal of the band crosses the horizontal axis. You can imagine the matrix as the fourth quadrant of the Cartesian coordinate system. Then the main diagonal of an alignment matrix is described by the function `f(x) = -x`, all diagonals that crosses the vertical axis below this point are specified with negative values while all diagonals that crosses the horizontal axis are specified with positive values

(see image). A given band is valid as long as the relation `lower diagonal <= upper diagonal` holds. In case of equality, the alignment is equivalent to the hamming distance problem, where only substitutions are considered.

Important: The alignment algorithms return `MinValue<ScoreValue>::VALUE` if a correct alignment cannot be computed due to invalid compositions of the band and the specified alignment preferences. Assume, you compute a global alignment and the given band does not cover the last cell of the alignment matrix. In this case it is not possible to compute a correct alignment, hence `MinValue<ScoreValue>::VALUE` is returned, while no further alignment information are computed.

Let's compute a banded alignment. The first step is to write the `main` function body including the type definitions and the initializations.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;                 // sequence type
    typedef Align<TSequence, ArrayGaps> TAlign;     // align type

    TSequence seq1 = "CDFGHC";
    TSequence seq2 = "CDEFGAHC";

    TAlign align;
    resize(rows(align), 2);
    assignSource(row(align, 0), seq1);
    assignSource(row(align, 1), seq2);
```

After we initialized everything, we will compute the banded alignment. We pass the values `-2` for the lower diagonal and `2` for the upper diagonal.

```
    int score = globalAlignment(align, Score<int, Simple>(0, -1, -1), -2, 2);
    std::cout << "Score: " << score << std::endl;
    std::cout << align << std::endl;

    return 0;
}
```

And here is the output:

```
Score: -2
      0     .
        CD-FG-HC
        || || ||
        CDEFGAHC
```

Assignment 5

Type Transfer

Objective Write an approximate pattern matching algorithm using alignment algorithms. Report the positions of all hits where the pattern matches the text with at most `2` errors. Output the number of total edits used to match the

pattern and print the corresponding cigar string of the alignment without leading and trailing gaps in the pattern. Text: "MISSISSIPPIANDMISSOURI" Pattern: "SISSI"

Hint

- The first step would be to verify at which positions in the text the pattern matches with at most 2 errors.
- Use the infix function to return a subsequence of a string.
- A CIGAR string is a different representation of an alignment. It consists of a number followed by an operation. The number indicates how many operations are executed. Operations can be **M** for match or mismatch, **I** for insertion and **D** for deletion. Here is an example:

```
ref: AC--GTCATTT
r01: ACGTCTCA---
Cigar of r01: 2M2I4M3D
```

Solution (Step 1)

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;

    TSequence text =    "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";

    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text, i, i + length(pattern));

        // Report hits with at most 2 errors.
        if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
        {
            appendValue(locations, i);
        }
    }
    return 0;
}
```

Solution (Step 2)

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text =    "MISSISSIPPIANDMISSOURI";
```

```
    TSequence pattern = "SISSI";

    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text, i, i + length(pattern));

        // Report hits with at most 2 errors.
        if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
        {
            appendValue(locations, i);
        }
    }

    TGaps gapsText;
    TGaps gapsPattern;
    assignSource(gapsPattern, pattern);
    std::cout << "Text: " << text << "\tPattern: " << pattern << std::endl;
    for (TIterator it = begin(locations); it != end(locations); ++it)
    {
        // Clear previously computed gaps.
        clearGaps(gapsText);
        clearGaps(gapsPattern);

        // Only recompute the area within the current window over the text.
        TSequence textInfix = infix(text, *it, *it + length(pattern));
        assignSource(gapsText, textInfix);

        // Use semi-global alignment since we do not want to track leading/
↪trailing gaps in the pattern.
        // Restirct search space using a band allowing at most 2 errors in␣
↪vertical/horizontal direction.
        int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -
↪1), AlignConfig<true, false, false, true>(), -2, 2);
        std::cout << "Hit at position " << *it << "\ttotal edits: " <<␣
↪abs(score) << std::endl;
    }

    return 0;
}
```

Solution (Step 3)

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text =    "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";
```

```
    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text, i, i + length(pattern));

        // Report hits with at most 2 errors.
        if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
        {
            appendValue(locations, i);
        }
    }

    TGaps gapsText;
    TGaps gapsPattern;
    assignSource(gapsPattern, pattern);
    std::cout << "Text: " << text << "\tPattern: " << pattern << std::endl;
    for (TIterator it = begin(locations); it != end(locations); ++it)
    {
        // Clear previously computed gaps.
        clearGaps(gapsText);
        clearGaps(gapsPattern);

        // Only recompute the area within the current window over the text.
        TSequence textInfix = infix(text, *it, *it + length(pattern));
        assignSource(gapsText, textInfix);

        // Use semi-global alignment since we do not want to track leading/
↪trailing gaps in the pattern.
        // Restirct search space using a band allowing at most 2 errors in␣
↪vertical/horizontal direction.
        int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -
↪1), AlignConfig<true, false, false, true>(), -2, 2);

        TGapsIterator itGapsPattern = begin(gapsPattern);
        TGapsIterator itGapsEnd = end(gapsPattern);

        // Remove trailing gaps in pattern.
        int count = 0;
        while (isGap(--itGapsEnd))
            ++count;
        setClippedEndPosition(gapsPattern, length(gapsPattern) - count);

        // Remove leading gaps in pattern.
        if (isGap(itGapsPattern))
        {
            setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
            setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
        }
        std::cout << "Hit at position " << *it << "\ttotal edits: " <<␣
↪abs(score) << std::endl;
    }
    return 0;
}
```

Solution (Step 4)

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text =    "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";

    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text, i, i + length(pattern));

        // Report hits with at most 2 errors.
        if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
        {
            appendValue(locations, i);
        }
    }

    TGaps gapsText;
    TGaps gapsPattern;
    assignSource(gapsPattern, pattern);
    std::cout << "Text: " << text << "\tPattern: " << pattern << std::endl;
    for (TIterator it = begin(locations); it != end(locations); ++it)
    {
        // Clear previously computed gaps.
        clearGaps(gapsText);
        clearGaps(gapsPattern);

        // Only recompute the area within the current window over the text.
        TSequence textInfix = infix(text, *it, *it + length(pattern));
        assignSource(gapsText, textInfix);

        // Use semi-global alignment since we do not want to track leading/
↪trailing gaps in the pattern.
        // Restirct search space using a band allowing at most 2 errors in␣
↪vertical/horizontal direction.
        int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -
↪1), AlignConfig<true, false, false, true>(), -2, 2);

        TGapsIterator itGapsPattern = begin(gapsPattern);
        TGapsIterator itGapsEnd = end(gapsPattern);

        // Remove trailing gaps in pattern.
        int count = 0;
        while (isGap(--itGapsEnd))
            ++count;
        setClippedEndPosition(gapsPattern, length(gapsPattern) - count);
```

```
        // Remove leading gaps in pattern.
        if (isGap(itGapsPattern))
        {
            setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
            setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
        }

        // Reinitilaize the iterators.
        TGapsIterator itGapsText = begin(gapsText);
        itGapsPattern = begin(gapsPattern);
        itGapsEnd = end(gapsPattern);

        // Use a stringstream to construct the cigar string.
        std::stringstream cigar;
        while (itGapsPattern != itGapsEnd)
        {
            // Count insertions.
            if (isGap(itGapsText))
            {
                int numGaps = countGaps(itGapsText);
                cigar << numGaps << "I";
                itGapsText += numGaps;
                itGapsPattern += numGaps;
                continue;
            }
            ++itGapsText;
            ++itGapsPattern;
        }
        std::cout << "Hit at position " << *it << "\ttotal edits: " <<␣
↪abs(score) << std::endl;
    }
    return 0;
}
```

Solution (Step 5)

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text =    "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";

    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text, i, i + length(pattern));

        // Report hits with at most 2 errors.
```

```
        if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
        {
            appendValue(locations, i);
        }
    }

    TGaps gapsText;
    TGaps gapsPattern;
    assignSource(gapsPattern, pattern);
    std::cout << "Text: " << text << "\tPattern: " << pattern << std::endl;
    for (TIterator it = begin(locations); it != end(locations); ++it)
    {
        // Clear previously computed gaps.
        clearGaps(gapsText);
        clearGaps(gapsPattern);

        // Only recompute the area within the current window over the text.
        TSequence textInfix = infix(text, *it, *it + length(pattern));
        assignSource(gapsText, textInfix);

        // Use semi-global alignment since we do not want to track leading/
↪trailing gaps in the pattern.
        // Restirct search space using a band allowing at most 2 errors in␣
↪vertical/horizontal direction.
        int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -
↪1), AlignConfig<true, false, false, true>(), -2, 2);

        TGapsIterator itGapsPattern = begin(gapsPattern);
        TGapsIterator itGapsEnd = end(gapsPattern);

        // Remove trailing gaps in pattern.
        int count = 0;
        while (isGap(--itGapsEnd))
            ++count;
        setClippedEndPosition(gapsPattern, length(gapsPattern) - count);

        // Remove leading gaps in pattern.
        if (isGap(itGapsPattern))
        {
            setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
            setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
        }

        // Reinitilaize the iterators.
        TGapsIterator itGapsText = begin(gapsText);
        itGapsPattern = begin(gapsPattern);
        itGapsEnd = end(gapsPattern);

        // Use a stringstream to construct the cigar string.
        std::stringstream cigar;
        while (itGapsPattern != itGapsEnd)
        {
            // Count insertions.
            if (isGap(itGapsText))
            {
                int numGaps = countGaps(itGapsText);
                cigar << numGaps << "I";
                itGapsText += numGaps;
```

```
                itGapsPattern += numGaps;
                continue;
            }
            // Count deletions.
            if (isGap(itGapsPattern))
            {
                int numGaps = countGaps(itGapsPattern);
                cigar << numGaps << "D";
                itGapsText += numGaps;
                itGapsPattern += numGaps;
                continue;
            }
            ++itGapsText;
            ++itGapsPattern;
        }
        // Output the hit position in the text, the total number of edits and␣
↪the corresponding cigar string.
        std::cout << "Hit at position  " << *it << "\ttotal edits: " <<␣
↪abs(score) << std::endl;
    }

    return 0;
}
```

Solution (Step 6)

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text =    "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";

    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text, i, i + length(pattern));

        // Report hits with at most 2 errors.
        if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
        {
            appendValue(locations, i);
        }
    }

    TGaps gapsText;
    TGaps gapsPattern;
    assignSource(gapsPattern, pattern);
    std::cout << "Text: " << text << "\tPattern: " << pattern << std::endl;
```

```
    for (TIterator it = begin(locations); it != end(locations); ++it)
    {
        // Clear previously computed gaps.
        clearGaps(gapsText);
        clearGaps(gapsPattern);

        // Only recompute the area within the current window over the text.
        TSequence textInfix = infix(text, *it, *it + length(pattern));
        assignSource(gapsText, textInfix);

        // Use semi-global alignment since we do not want to track leading/
↪trailing gaps in the pattern.
        // Restirct search space using a band allowing at most 2 errors in␣
↪vertical/horizontal direction.
        int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -
↪1), AlignConfig<true, false, false, true>(), -2, 2);

        TGapsIterator itGapsPattern = begin(gapsPattern);
        TGapsIterator itGapsEnd = end(gapsPattern);

        // Remove trailing gaps in pattern.
        int count = 0;
        while (isGap(--itGapsEnd))
            ++count;
        setClippedEndPosition(gapsPattern, length(gapsPattern) - count);

        // Remove leading gaps in pattern.
        if (isGap(itGapsPattern))
        {
            setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
            setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
        }

        // Reinitilaize the iterators.
        TGapsIterator itGapsText = begin(gapsText);
        itGapsPattern = begin(gapsPattern);
        itGapsEnd = end(gapsPattern);

        // Use a stringstream to construct the cigar string.
        std::stringstream cigar;
        int numMatchAndMismatches = 0;
        while (itGapsPattern != itGapsEnd)
        {
            // Count insertions.
            if (isGap(itGapsText))
            {
                int numGaps = countGaps(itGapsText);
                cigar << numGaps << "I";
                itGapsText += numGaps;
                itGapsPattern += numGaps;
                continue;
            }
            // Count deletions.
            if (isGap(itGapsPattern))
            {
                int numGaps = countGaps(itGapsPattern);
                cigar << numGaps << "D";
                itGapsText += numGaps;
```

```
                itGapsPattern += numGaps;
                continue;
            }

            // Count matches and  mismatches.
            while (itGapsPattern != itGapsEnd)
            {
                if (isGap(itGapsPattern) || isGap(itGapsText))
                   break;

                ++numMatchAndMismatches;
                ++itGapsText;
                ++itGapsPattern;
            }
            if (numMatchAndMismatches != 0)
                cigar << numMatchAndMismatches << "M";
            numMatchAndMismatches = 0;
        }
        // Output the hit position in the text, the total number of edits and␣
↪the corresponding cigar string.
        std::cout << "Hit at position  " << *it << "\ttotal edits: " <<␣
↪abs(score) << "\tcigar: " << cigar.str() << std::endl;
    }

    return 0;
}
```

Complete Solution (and more explanations) Write the *main* body of the program with type definition and initalization of the used data structures.

```
#include <iostream>
#include <seqan/align.h>

using namespace seqan;

int main()
{
    typedef String<char> TSequence;
    typedef Gaps<TSequence, ArrayGaps> TGaps;
    typedef Iterator<TGaps>::Type TGapsIterator;
    typedef Iterator<String<int> >::Type TIterator;

    TSequence text =    "MISSISSIPPIANDMISSOURI";
    TSequence pattern = "SISSI";
```

In the first part of the algorithm we implement an alignment based verification process to identify positions in the *subject sequence* at which we can find our pattern with at most *2* errors. We slide the *5*5* alignment matrix position by position over the *subject sequence* and use the *MeyersBitVector* to verify the hits. If the score is greater or equal than *-2*, then we have found a hit. We store the begin position of the hit in *locations*.

```
    String<int> locations;
    for (unsigned i = 0; i < length(text) - length(pattern); ++i)
    {
        // Compute the MyersBitVector in current window of text.
        TSequence tmp = infix(text, i, i + length(pattern));
```

```
        // Report hits with at most 2 errors.
        if (globalAlignmentScore(tmp, pattern, MyersBitVector()) >= -2)
        {
            appendValue(locations, i);
        }
    }
```

In the second part of the algorithm we iterate over all reported locations. This time we compute a semi-global alignment since we won't penalize gaps at the beginning and at the end of our pattern. We also compute a band allowing at most *2* errors in either direction. Don't forget to clear the gaps in each iteration, otherwise we might encounter wrong alignments.

```
    TGaps gapsText;
    TGaps gapsPattern;
    assignSource(gapsPattern, pattern);
    std::cout << "Text: " << text << "\tPattern: " << pattern << std::endl;
    for (TIterator it = begin(locations); it != end(locations); ++it)
    {
        // Clear previously computed gaps.
        clearGaps(gapsText);
        clearGaps(gapsPattern);
        // Only recompute the area within the current window over the text.
        TSequence textInfix = infix(text, *it, *it + length(pattern));
        assignSource(gapsText, textInfix);

        // Use semi-global alignment since we do not want to track leading/
↪trailing gaps in the pattern.
        // Restirct search space using a band allowing at most 2 errors in␣
↪vertical/horizontal direction.
        int score = globalAlignment(gapsText, gapsPattern, Score<int>(0, -1, -
↪1), AlignConfig<true, false, false, true>(), -2, 2);
```

In the next part we determine the cigar string for the matched pattern. We have to remove leading and trailing gaps in the *gapsPattern* object using the functions setClippedBeginPosition and setClippedEndPosition. We also need to set the clipped begin position for the *gapsText* object such that both Gaps begin at the same position.

```
        TGapsIterator itGapsPattern = begin(gapsPattern);
        TGapsIterator itGapsEnd = end(gapsPattern);

        // Remove trailing gaps in pattern.
        int count = 0;
        while (isGap(--itGapsEnd))
            ++count;
        setClippedEndPosition(gapsPattern, length(gapsPattern) - count);

        // Remove leading gaps in pattern.
        if (isGap(itGapsPattern))
        {
            setClippedBeginPosition(gapsPattern, countGaps(itGapsPattern));
            setClippedBeginPosition(gapsText, countGaps(itGapsPattern));
        }

        // Reinitilaize the iterators.
        TGapsIterator itGapsText = begin(gapsText);
        itGapsPattern = begin(gapsPattern);
        itGapsEnd = end(gapsPattern);
```

```
        // Use a stringstream to construct the cigar string.
        std::stringstream cigar;
        int numMatchAndMismatches = 0;
        while (itGapsPattern != itGapsEnd)
        {
```

First, we identify insertions using the functions isGap and countGaps.

```
            // Count insertions.
            if (isGap(itGapsText))
            {
                int numGaps = countGaps(itGapsText);
                cigar << numGaps << "I";
                itGapsText += numGaps;
                itGapsPattern += numGaps;
                continue;
            }
```

We do the same to identify deletions.

```
            // Count deletions.
            if (isGap(itGapsPattern))
            {
                int numGaps = countGaps(itGapsPattern);
                cigar << numGaps << "D";
                itGapsText += numGaps;
                itGapsPattern += numGaps;
                continue;
            }
```

If there is neither an insertion nor a deletion, then there must be a match or a mismatch. As long as we encounter matches and mismatches we move forward in the Gaps structures. Finally we print out the position of the hit, its total number of edits and the corresponding cigar string.

```
            // Count matches and  mismatches.
            while (itGapsPattern != itGapsEnd)
            {
                if (isGap(itGapsPattern) || isGap(itGapsText))
                   break;

                ++numMatchAndMismatches;
                ++itGapsText;
                ++itGapsPattern;
            }
            if (numMatchAndMismatches != 0)
                cigar << numMatchAndMismatches << "M";
            numMatchAndMismatches = 0;
        }
        // Output the hit position in the text, the total number of edits and␣
↪the corresponding cigar string.
        std::cout << "Hit at position  " << *it << "\ttotal edits: " <<␣
↪abs(score) << "\tcigar: " << cigar.str() << std::endl;
    }

    return 0;
}
```

Here is the output of this program.

```
Text: MISSISSIPPIANDMISSOURI        Pattern: SISSI
Hit at position  0        total edits: 1        cigar: 5M
Hit at position  1        total edits: 1        cigar: 1I4M
Hit at position  2        total edits: 1        cigar: 4M1I
Hit at position  3        total edits: 0        cigar: 5M
Hit at position  4        total edits: 1        cigar: 1I4M
Hit at position  6        total edits: 2        cigar: 5M
Hit at position  14        total edits: 2        cigar: 4M1I
```

ToC

Contents

Multiple Sequence Alignment

Learning Objective You will learn how to compute a multiple sequence alignment (MSA) using SeqAn's alignment data structures and algorithms.

Difficulty Basic

Duration 30 min

Prerequisites *Sequences*, *Alignment*

Alignments are at the core of biological sequence analysis and part of the "bread and butter" tasks in this area. As you have learned in the *pairwise alignment tutorial*, SeqAn offers powerful and flexible functionality for coputing such pairwise alignments. This tutorial shows how to compute multiple sequence alignments (MSAs) using SeqAn. First, some background on MSA will be given and the tutorial will then explain how to create multiple sequence alignments.

Note that this tutorial focuses on the `<seqan/graph_msa.h>` module whose purpose is the computation of **global** MSAs, i.e. similar to SeqAn::T-Coffe *[REW+08]* or ClustalW *[THG94]*. If you are interested in computing consensus sequences of multiple overlapping sequences (e.g. NGS reads), similar to assembly after the layouting step, then have a look at the *Consensus Alignment* tutorial.

While the pairwise alignment of sequences can be computed exactly in quadratic time usind dynamic programming, the computation of exact MSAs is harder. Given n sequences of length ℓ, the exact computation of an MSA is only feasible in time $\mathcal{O}(\ell^n)$. Thus, global MSAs are usually computed using a heuristic called **progressive alignment**. For an introduction to MSAs, see the Wikipedia Article on Multiple Sequence Aligment.

Computing MSAs with SeqAn

The SeqAn library gives you access to the engine of SeqAn::T-Coffee *[REW+08]*, a powerful and efficient MSA algorithm based on the progressive alignment strategy. The easiest way to compute multiple sequence alignments is using the function globalMsaAlignment. The following example shows how to compute a global multiple sequence alignment of proteins using the Blosum62 scoring matrix with gap extension penalty `-11` and gap open penalty `-1`.

First, we include the necessary headers and begin the `main` function by declaring our strings as a char array.

```
#include <iostream>
#include <seqan/align.h>
#include <seqan/graph_msa.h>

using namespace seqan;

int main()
{
    char const * strings[4] =
    {
        "DPKKPRGKMSSYAFFVQTSREEHKKKHPDASVNFSEFSKKCSERWKTMSAKEKGKFEDMA"
        "KADKARYEREMKTYIPPKGE",
        "RVKRPMNAFIVWSRDQRRKMALENPRMRNSEISKQLGYQWKMLTEAEKWPFFQEAQKLQA"
        "MHREKYPNYKYRPRRKAKMLPK",
        "FPKKPLTPYFRFFMEKRAKYAKLHPEMSNLDLTKILSKKYKELPEKKKMKYIQDFQREKQ"
        "EFERNLARFREDHPDLIQNAKK",
        "HIKKPLNAFMLYMKEMRANVVAESTLKESAAINQILGRRWHALSREEQAKYYELARKERQ"
        "LHMQLYPGWSARDNYGKKKKRKREK"
    };
```

Next, we build a Align object with underling SeqAn Strings over the AminoAcid alphabet. We create four rows and assign the previously defined amino acid strings into the rows.

```
    Align<String<AminoAcid> > align;
    resize(rows(align), 4);
    for (int i = 0; i < 4; ++i)
        assignSource(row(align, i), strings[i]);
```

Finally, we call globalMsaAlignment and print `align` to the standard output. We use the Blosum62 score matrix with the penalties from above.

```
    globalMsaAlignment(align, Blosum62(-1, -11));
    std::cout << align << "\n";

    return 0;
}
```

The output of the program look as follows.

```
      0     .    :    .    :    .    :    .    :    .    :
        DPKKPRGKMSSYAFFVQTSREEHKKKHPDASVNFSEFSKKCSERWKTMSA
          | |   |           |        |    |  | ||     ||
        RVKRP---MNAFIVWSRDQRRKMALENPRMR-NS-EISKQLGYQWKMLTE
          | |               | | |    | |  |     | |    | | |
        FPKKP---LTPYFRFFMEKRAKYAKLHPEMS-NL-DLTKILSKKYKELPE
          |||   |         | ||                   ||      |
        HIKKP---LNAFMLYMKEMRANVVAESTLKE-SA-AINQILGRRWHALSR

     50     .    :    .    :    .    :    .    :
        KEKGKFEDMAKADKARYEREMKTY----------IPPKGE
         ||  |   |    |         |
        AEKWPFFQEAQKLQAMHREKYPNY--KYRP-RRKAKMLPK
          |    |  |                            |
        KKKMKYIQDFQREKQEFERNLARF--REDH-PDLIQNAKK
            ||      | |           |             |
        EEQAKYYELARKERQLHMQLYPGWSARDNYGKKKKRKREK
```

Note that we stored the MSA in an Align object which allows easy access to the individual rows of the MSA as Gaps

objects. globalMsaAlignment also allows storing the alignment as an AlignmentGraph. While this data structure makes other operations easier, it is less intuitive than the tabular represention of the Align class.

Assignment 1

Type Review

Objective Compute a multiple sequence alignments between the four protein sequences from above using a Align object and the Blosum80 score matrix.

Solution The solution looks as follows.

```
//![main]
#include <iostream>
#include <seqan/align.h>
#include <seqan/graph_msa.h>

using namespace seqan;

int main()
{
    char const * strings[4] =
    {
        "DPKKPRGKMSSYAFFVQTSREEHKKKHPDASVNFSEFSKKCSERWKTMSAKEKGKFEDMA"
        "KADKARYEREMKTYIPPKGE",
        "RVKRPMNAFIVWSRDQRRKMALENPRMRNSEISKQLGYQWKMLTEAEKWPFFQEAQKLQA"
        "MHREKYPNYKYRPRRKAKMLPK",
        "FPKKPLTPYFRFFMEKRAKYAKLHPEMSNLDLTKILSKKYKELPEKKKMKYIQDFQREKQ"
        "EFERNLARFREDHPDLIQNAKK",
        "HIKKPLNAFMLYMKEMRANVVAESTLKESAAINQILGRRWHALSREEQAKYYELARKERQ"
        "LHMQLYPGWSARDNYGKKKKRKREK"
    };

    Align<String<AminoAcid> > align;
    resize(rows(align), 4);
    for (int i = 0; i < 4; ++i)
        assignSource(row(align, i), strings[i]);

    globalMsaAlignment(align, Blosum80(-1, -11));
    std::cout << align << "\n";

    return 0;
}
//![main]
```

And here is the program's output.

```
      0     .    :    .    :    .    :    .    :    .    :
        DPKKPRGKMSSYAFFVQTSREEHKKKHPDASVNFSEFSKKCSERWKTMSA
          | |   |           |        |    |  | ||     ||
        RVKRP---MNAFIVWSRDQRRKMALENPRMR-NS-EISKQLGYQWKMLTE
          | |               | | |   | |  |     | |    | | |
        FPKKP---LTPYFRFFMEKRAKYAKLHPEMS-NL-DLTKILSKKYKELPE
          |||   |         | ||                   ||      |
        HIKKP---LNAFMLYMKEMRANVVAESTLKE-SA-AINQILGRRWHALSR

     50     .    :    .    :    .    :    .    :    .    :
        KEKGKFEDMAKADKARYEREMKTY---------------IP--PKG---E
```

```
          ||  |   |    |   || |                   |
        AEKWPFFQEAQKLQAMH-RE-K-----YP------NYKYRPRRKAKMLPK
          |    |  |       |         |               ||   |
        KKKMKYIQDFQREKQEFERNLARFREDHP------DL--IQ--NAK---K
            ||      | |             |                   |
        EEQAKYYELARKERQLH-MQ-L-----YPGWSARDNYGKKKKRKRE---K
```

Computing Consensus Sequences

One common task following the computation of a global MSA for DNA sequences is the computation of a consensus sequence. The type ProfileChar can be used for storing counts for a profile's individual characters. It is used by creating a String over ProfileChar as the alphabet.

The following program first computes a global MSA of four variants of exon1 of the gene SHH. First, we compute the alignment as in the example above.

```
#include <iostream>
#include <seqan/align.h>
#include <seqan/graph_msa.h>

using namespace seqan;

int main()
{
    // some variangs of sonic hedgehog exon 1
    char const * strings[4] =
    {
        // gi|2440284|dbj|AB007129.1| Oryzias latipes
        "GCGGGTCACTGAGGGCTGGGATGAGGACGGCCACCACTTCGAGGAGTCCCTTCACTACGAGGGCAGGGCC"
        "GTGGACATCACCACGTCAGACAGGGACAAGAGCAAGTACGGCACCCTGTCCAGACTGGCGGTGGAAGCTG"
        "GGTTCGACTGGGTCTACTATGAGTCCAAAGCGCACATCCACTGCTCTGTGAAAGCAGAAAGCTCAGTCGC"
        "TGCAAAGTCGGGCGGTTGCTTCCCAGGATCCTCCACGGTCACCCTGGAAAATGGCACCCAGAGGCCCGTC"
        "AAAGATCTCCAACCCGGGGACAGAGTACTGGCCGCGGATTACGACGGAAACCCGGTTTATACCGACTTCA"
        "TCATGTTCAA",
        // gi|1731488|gb|U51350.1|DDU51350 Devario devario
        "CTACGGCAGAAGAAGACATCCGAAAAAGCTGACACCTCTCGCCTACAAGCAGTTCATACCTAATGTCGCG"
        "GAGAAGACCTTAGGGGCCAGCGGCAGATACGAGGGCAAGATAACGCGCAATTCGGAGAGATTTAAAGAAC"
        "TTACTCCAAATTACAATCCCGACATTATCTTTAAGGATGAGGAGAACACG",
        // gi|1731504|gb|U51352.1|PTU51352 Puntius tetrazona
        "CTACGGCAGAAGAAGACATCCCAAGAAGCTGACACCTCTCGCCTACAAGCAGTTTATACCTAATGTCGCG"
        "GAGAAGACCTTAGGGGCCAGCGGCAGATACGAGGGCAAGATCACGCGCAATTCGGAGAGATTTAAAGAAC"
        "TTACTCCAAATTACAATCCCGACATTATCTTTAAGGATGAGGAGAACACT",
        // gi|54399708|gb|AY642858.1| Bos taurus
        "TGCTGCTGCTGGCGAGATGTCTGCTGGTGCTGCTTGTCTCCTCGCTGTTGATGTGCTCGGGGCTGGCGTG"
        "CGGACCCGGCAGGGGATTTGGCAAGAGGCGGAACCCCAAAAAGCTGACCCCTTTAGCCTACAAGCAGTTT"
        "ATCCCCAACGTGGCGGAGAAGACCCTAGGGGCCAGTGGAAGATATGAGGGGAAGATCACCAGAAACTCAG"
        "AGCGATTTAAGGAACTCACCCCCAATTACAACCC"
    };

    Align<DnaString> align;
    resize(rows(align), 4);
    for (int i = 0; i < 4; ++i)
        assignSource(row(align, i), strings[i]);

    globalMsaAlignment(align, SimpleScore(5, -3, -1, -3));
    std::cout << align << "\n";
```

Then, we create the profile string with the length of the MSA. We then count the number of characters (and gap pseudo-characters which have an `ordValue` of `4` for Gaps over Dna) at each position.

```
    // create the profile string
    String<ProfileChar<Dna> > profile;
    resize(profile, length(row(align, 0)));
    for (unsigned rowNo = 0; rowNo < 4u; ++rowNo)
        for (unsigned i = 0; i < length(row(align, rowNo)); ++i)
            profile[i].count[ordValue(getValue(row(align, rowNo), i))] += 1;
```

Finally, we compute the consensus and print it to the standard output. At each position, the consensus is called as the character with the highest count. Note that `getMaxIndex` breaks ties by the ordinal value of the caracters, i.e. `A` would be preferred over `C`, `C` over `G` and so on.

```
    // call consensus from this string
    DnaString consensus;
    for (unsigned i = 0; i < length(profile); ++i)
    {
        int idx = getMaxIndex(profile[i]);
        if (idx < 4)  // is not gap
            appendValue(consensus, Dna(getMaxIndex(profile[i])));
    }

    std::cout << "consensus sequence is\n"
              << consensus << "\n";

    return 0;
}
```

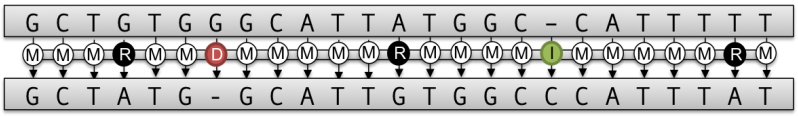
The output of the program is as follows.

```
      0     .    :    .    :    .    :    .    :    .    :
        -GCGGGTCACTGAG-GGCTGGGATGA-----------------------
          |   | ||     |||    | ||
        --C---T-AC-----GGC----A-GA-----------------------
          |   | ||     |||    | ||
        --C---T-AC-----GGC----A-GA-----------------------
          |   |  |     |||    | ||
        TGC---T-GCT-GCTGGC---GA-GATGTCTGCTGGTGCTGCTTGTCTCC

     50     .    :    .    :    .    :    .    :    .    :
        ---------------------------------------GG---------
                                                |
        ---------------------------------------AG---------
                                                ||
        ---------------------------------------AG---------
                                                ||
        TCGCTGTTGATGTGCTCGGGGCTGGCGTGCGGACCCGGCAGGGGATTTGG

    100     .    :    .    :    .    :    .    :    .    :
        -ACGGCCAC--C---ACTTCGAGGAGTCCCTTCACTACGAGGGCAGGGCC
         | |   |   |   |          |||    |  |  |    |  ||
        -AAG---A---C---A----------TCC---GA--A-AA----A--GC-
         |||   |   |   |          |||    |  |  |    |  ||
        -AAG---A---C---A----------TCC---CA--A-GA----A--GC-
         |||   |   |   |           ||   ||  |  |    |  ||
        CAAG---A-GGCGGAA----------CCC---CA--A-AA----A--GC-
```

```
    150     .    :    .    :    .    :    .    :    .    :
        GTGGACATCACCACGTCAGACAGGGACAAGAGCAAG--TACGGCACCCTG
         | ||   |||| | || | |    |||  ||| ||   |    | |||
        -T-GA---CACCTC-TC-GCC---TACA--AGC-AGTTCA---TA-CCT-
         | ||   |||||| || |||   ||||  ||| |||| |   || |||
        -T-GA---CACCTC-TC-GCC---TACA--AGC-AGTTTA---TA-CCT-
         | ||   | |||  |  |||   ||||  ||| ||||||   |  ||
        -T-GA---CCCCTT-TA-GCC---TACA--AGC-AGTTTA---TC-CCC-

    200     .    :    .    :    .    :    .    :    .    :
        TCCAGACTG--GCGGTGGAAGCTGGGTTCGACTGGGTCTACTATGAGTCC
           | | ||  ||||  |||         |||     ||  || | | ||
        ---A-A-TGTCGCGG-AGAA---------GAC-----CT--TA-GGGGCC
           | | |||||||| ||||         |||     ||  || ||||||
        ---A-A-TGTCGCGG-AGAA---------GAC-----CT--TA-GGGGCC
           | |  || |||| ||||         |||     |   || ||||||
        ---A-A-CGTGGCGG-AGAA---------GAC-----CC--TA-GGGGCC

    250     .    :    .    :    .    :    .    :    .    :
        AAAGC-GCACATCCACTGCTCTGTGAAAGCAGAAAGCTCAGTCGCT-GCA
          ||| ||| ||  || |    | |   ||   ||| | |  |||  ||
        --AGCGGCAGAT--AC-G---AG-G---GC---AAGATAA--CGC--GC-
          ||||||||||  || |   || |   ||   ||||| |  |||  ||
        --AGCGGCAGAT--AC-G---AG-G---GC---AAGATCA--CGC--GC-
          || || ||||  |  |   || |   |    |||||||  | |  |
        --AGTGGAAGAT--AT-G---AG-G---GG---AAGATCA--C-C-AGA-

    300     .    :    .    :    .    :    .    :    .    :
        AAGTCGGGCGGTTGCTTCCCAG-GATCCTCCACGGTCACCCTGGAAAATG
        || ||||             || |||  |      |        |||
        AATTCGG-------------AGAGAT--T------T--------AAA---
        |||||||             ||||||  |      |        |||
        AATTCGG-------------AGAGAT--T------T--------AAA---
        || || |             || |||  |      |        ||
        AACTCAG-------------AGCGAT--T------T--------AAG---

    350     .    :    .    :    .    :    .    :    .    :
        GCACCCAGAGGCCCGTCAAAGATCTCCAA-CCCGGGGACAGAGTACTGGC
        | ||  |    | |  |||  || | ||| |||   |||  | || |
        GAACTTA----CTC--CAA--AT-TACAATCCC---GAC--ATTA-T---
        |||||||    |||  |||  || |||||||||   |||  |||| |
        GAACTTA----CTC--CAA--AT-TACAATCCC---GAC--ATTA-T---
        ||||| |    | |  | |  || ||||| |||
        GAACTCA----CCC--CCA--AT-TACAA-CCC-----------------

    400     .    :    .    :    .    :    .    :    .    :
        CGCGGA-TTACGACGGA--AACCCGGTTTATACCGACTTCATCATGTTCA
          |    |||  | |||  |    ||   |    ||    | || |
        --C---TTTA--A-GGATGA----GG---A----GA----A-CACG----
          |   ||||  | ||||||    ||   |    ||    | |||
        --C---TTTA--A-GGATGA----GG---A----GA----A-CACT----

        --------------------------------------------------

    450
        A

        -
```

```
        _

        _



consensus sequence is
GCTACTGGCGAGAAGAAGACATCCCAAAAAGCTGACACCTCTCGCCTACAAGCAGTTTATACCTAATGTCGCGGAGAAGACCTTAGGGGCCAGCGGCAGAT
```

In bioinformatics, we often search for similarities between sequences. You might know the problem already from the pattern matching section. However, when we want to put a sequence in its evolutionary context, a simple pattern matching algorithm will not suffice, as it rather looks for small identical parts with some errors. But, if we have two sequences at hand, we don't actually know which patterns we are looking for.



To solve this problem, one computes a global pairwise sequence alignment to obtain an optimal transcript, that describes how these two sequences are related to each other. The transcript describes the edit operations (match, substitution, insertion and deletion) necessary to translate the one sequence into the other, as can be seen in the picture above.

The alignment problem is solved with a dynamic programming (DP) algorithm which runs in $\mathcal{O}(n^2)$ time and space. Besides the global alignment approach many, many more variations of this DP based algorithm have been developed over time. SeqAn unified all of these approaches into a single DP core implementation which can be extended pretty easily and thus with all possible configurations is a very versatile and powerful tool to compute many desired alignment variants.

In the *pairwise sequence alignment tutorial* you will learn more about the different alignment algorithms that are implemented and how to compute them.

A more complex problem then the pairwise sequence alignment is the multiple sequence alignment, where the optimal alignment between many sequences is sought. Solving the problem exactly would be infeasible, since the combinations would explode. In fact the runtime is $\mathcal{O}(n^k)$, where k is the number of sequences. Instead, one rather builds up a multiple sequence alignment by progressively computing pairwise alignments. The *multiple sequence alignment tutorial* teaches you how to compute a multiple sequence alignment in SeqAn.

ToC

Contents

Consensus Alignment

Learning Objective You will learn how to perform a consensus alignment of sequences (e.g. NGS reads) stored in a FragmentStore. After completing this tutorial, you will be able to perform a consensus alignment of reads with and without using alignment information. This is useful for the consensus step in sequence assembly.

Difficulty Advanced

Duration 1 h

Prerequisites *Fragment Store*

The SeqAn module <seqan/consensus.h> allows the computation of consensus alignments based on the method by Rausch et al. *[RKD+09]*. It can be used for the consensus step in Overlap-Layout-Consensus assemblers.

Consensus with Approximate Positions

The consensus module has two modes. The first one is applicable when approximate positions of the reads are known. The following program demonstrates this functionality.

First, we include the necessary headers.

```
#include <iostream>

#include <seqan/store.h>
#include <seqan/consensus.h>

using namespace seqan;

int main()
{
```

Next, the fragment store is filled with reads and approximate positions. The true alignment is shown in the comments.

```
    FragmentStore<> store;
    // Resize contigStore and contigNameStore (required for printing the first␣
↪layout).
    resize(store.contigStore, 1);
    appendValue(store.contigNameStore, "ref");

    // Actual read layout.
    //
    // AATGGATGGCAAAATAGTTGTTCCATGAATACATCTCTAAAGAGCTTT
    //           AAAATAGTTGTTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTT
    //                ␣
↪AGTTGTTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGTA
    //                                ACATCTCTAAAGAGCTTTGATGCTAATTTAGTCAAATT
    //                                                ␣
↪AGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGTACAATCTTCTCTAG

    // Append reads (includes small errors).
    appendRead(store, "AATGGATGGCAAAATAGTTGTTCCATGAATACATCTCTAAAGAGCTTT");
    appendRead(store, "AAAGTAGTTGTTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTT");
    appendRead(store, "AGTTGTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTTAGTCAATTTTCAATACTGTA
↪");
    appendRead(store, "ACATCTCTTAAAGAGCTTTGATGCTAATTTAGTCAAATT");
    appendRead(store, "AGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGTACAATCTTCTCTAG");

    // The position used in the following are only approximate and would
    // not lead to the read layout above.
    appendAlignedRead(store, 0, 0, 0, (int)length(store.readSeqStore[0]));
    appendAlignedRead(store, 1, 0, 12, 12 + (int)length(store.readSeqStore[1]));
    appendAlignedRead(store, 2, 0, 14, 14 + (int)length(store.readSeqStore[2]));
    appendAlignedRead(store, 3, 0, 18, 18 + (int)length(store.readSeqStore[3]));
    appendAlignedRead(store, 4, 0, 25, 25 + (int)length(store.readSeqStore[4]));
```

```
    // Print the (wrong) alignment.
    std::cout << "Initial alignment\n\n";
    AlignedReadLayout layout;
    layoutAlignment(layout, store);
    printAlignment(std::cout, layout, store, /*contigID=*/ 0, /*beginPos=*/ 0, /
↪*endPos=*/ 80, 0, 30);
```

This is followed by computing the consensus alignment using the function consensusAlignment.

```
    ConsensusAlignmentOptions options;
    options.useContigID = true;
    consensusAlignment(store, options);
```

Finally, the alignment is printed using an AlignedReadLayout object.

```
    std::cout << "Final alignment\n\n";
    layoutAlignment(layout, store);
    printAlignment(std::cout, layout, store, /*contigID=*/ 0, /*beginPos=*/ 0, /
↪*endPos=*/ 80, 0, 30);

    return 0;
}
```

Here is the program's output:

```
Initial alignment

--------------------------------------------------------------------------------
AATGGATGGCAAAATAGTTGTTCCATGAATACATCTCTAAAGAGCTTT
            AAAGTAGTTGTTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTT
              AGTTGTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTTAGTCAATTTTCAATACTGTA
                  ACATCTCTTAAAGAGCTTTGATGCTAATTTAGTCAAATT
                         AGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGTACAATCTTCTCTAG
Final alignment

AATGGATGGCAAAATAGTTGTTCCATGAATACATCTC-TAAAGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGT
.....................................*...........
          ...G.......................*......................
               .....*................*.........................*...............
                              .......T..............................
                                         .......................................
```

Consensus without Approximate Positions

When setting the `useContigID` member of the ConsensusAlignmentOptions object to `false` then we can also omit adding approximate positions for the reads. In this case, the consensus step performs an all-to-all alignment of all reads and then computes a consensus multi-read alignment for all of them. This is demonstrated by the following program.

```
#include <iostream>

#include <seqan/store.h>
#include <seqan/consensus.h>

using namespace seqan;
```

```
int main()
{
    FragmentStore<> store;

    // Actual read layout.
    //
    // AATGGATGGCAAAATAGTTGTTCCATGAATACATCTCTAAAGAGCTTT
    //           AAAATAGTTGTTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTT
    //                                                              
↪AGTTGTTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGTA
    //                                      ACATCTCTAAAGAGCTTTGATGCTAATTTAGTCAAATT
    //                                                              
↪AGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGTACAATCTTCTCTAG

    // Append reads (includes small errors).
    appendRead(store, "AATGGATGGCAAAATAGTTGTTCCATGAATACATCTCTAAAGAGCTTT");
    appendRead(store, "AAAGTAGTTGTTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTT");
    appendRead(store, "AGTTGTCCATGAATACATCTCTAAAGAGCTTTGATGCTAATTTAGTCAATTTTCAATACTGTA
↪");
    appendRead(store, "ACATCTCTTAAAGAGCTTTGATGCTAATTTAGTCAAATT");
    appendRead(store, "AGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGTACAATCTTCTCTAG");

    ConsensusAlignmentOptions options;
    options.useContigID = false;
    consensusAlignment(store, options);

    std::cout << "Final alignment\n\n";
    AlignedReadLayout layout;
    layoutAlignment(layout, store);
    printAlignment(std::cout, layout, store, /*contigID=*/ 0, /*beginPos=*/ 0, /
↪*endPos=*/ 80, 0, 30);

    return 0;
}
```

Here is this modified programs' output:

```
Final alignment

AATGGATGGCAAAATAGTTGTTCCATGAATACATCTC-TAAAGAGCTTTGATGCTAATTTAGTCAAATTTTCAATACTGT
.....................................*...........
          ...G.......................*.....................
               .....*................*.........................*..............
                              .......T..............................
                                         .......................................
```

ToC

Contents

Realignment

Learning Objective In this tutorial, you will learn how to refine multi-sequence alignments in a fragment store. This can be useful for refining multi-read alignments around indels prior to small variant calling. After completing the tutorial, you will be able to load reads into a fragment store and compute a realignment.

Difficulty Advanced

Duration 30 min

Prerequisites *Fragment Store*

A common task in NGS data analysis is small variant calling (SNVs or indels with a length of up to 10 bp) after the read mapping step. Usually, one considers the "pileup" of the reads and looks for variant signatures (e.g. a certain number of non-reference characters in the aligned reads). Usually, read mappers compute pairwise alignments of each read and the reference and store them in a SAM or BAM file. In the absence of indels, such pairwise alignments can be converted to a multi-read alignment without problems. However, there can be an undesired multi-read alignment around indels (Figure 1).

The task of improving such an alignment is called **realignment** and there is a small number of algorithms and tools available for realignment. This tutorial describes the `<seqan/realign.h>` module which implements a variant of the ReAligner algorithm by Anson and Myers *[AM97]*.

Getting Started

Consider the following program. It creates a fragment store and then reads a small reference (with a length of 2kb) from a FASTA file and also a SAM file with reads spanning a complex indel region at 1060 ~ 1140. Finally, it prints the multi-read alignment around this position using AlignedReadLayout.

```
#include <seqan/store.h>
#include <seqan/realign.h>

using namespace seqan;

int main()
{
    // Build paths to reference and to alignment.
    std::string refPath = getAbsolutePath("demos/tutorial/realignment/ref.fa");
    std::string samPath = getAbsolutePath("demos/tutorial/realignment/reads.sam");

    // Declare fragment store.
    FragmentStore<> store;

    // Load contigs and read alignment.
    loadContigs(store, refPath.c_str());
    BamFileIn bamFileIn(samPath.c_str());
    readRecords(store, bamFileIn);

    // Layout alignment and print.
    AlignedReadLayout layout;
    layoutAlignment(layout, store);
    printAlignment(std::cout, layout, store, /*contigID=*/ 0, /*posBegin=*/ 1060,
                   /*posEnd=*/ 1140, /*lineBegin=*/ 0, /*lineEnd=*/ 100);

    return 0;
}
```

The output of the program is as follows:

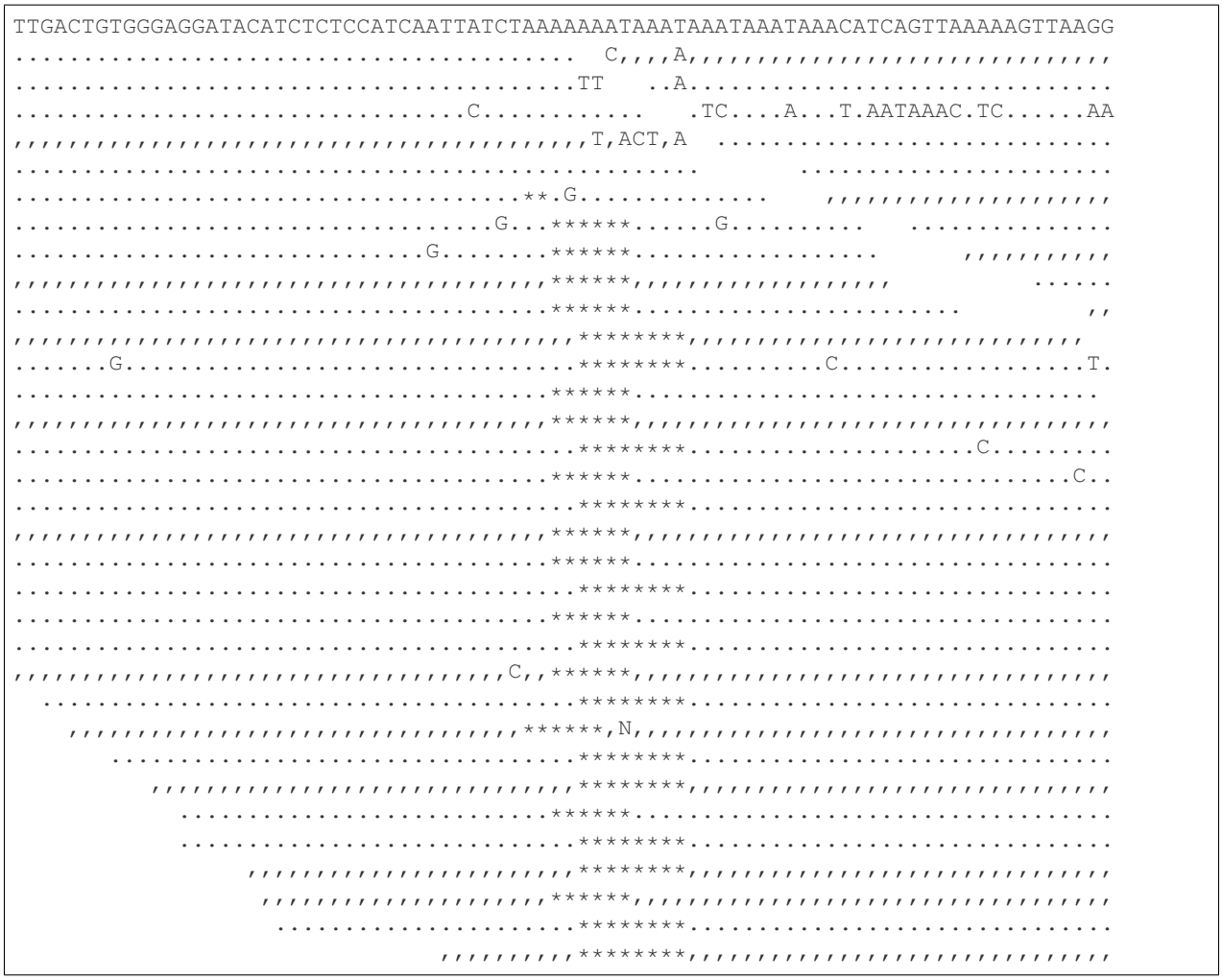


Figure 1: An example of a multi-read alignment from pairwise alignments

Performing the Realignment

We can now use the function reAlignment for performing a realignment of the reads in the fragment store.

contigID The numeric ID of the contig to realign.

realignmentMethod Whether to use linear (0) or affine gap costs (1). It is recommended to use affine gap costs.

bandwidth The bandwidth to use in the realignment step.

includeReference Whether or not to include the reference as a pseudo read.

The algorithm works as follows: A profile is computed, with a count of each base and the gap character at each position in the multi-read alignment. Each read is taken and aligned against this profile. This is repeated until convergence. Finally, the consensus of the multi-read alignment is written into `store.contigStore[contigID].seq`.

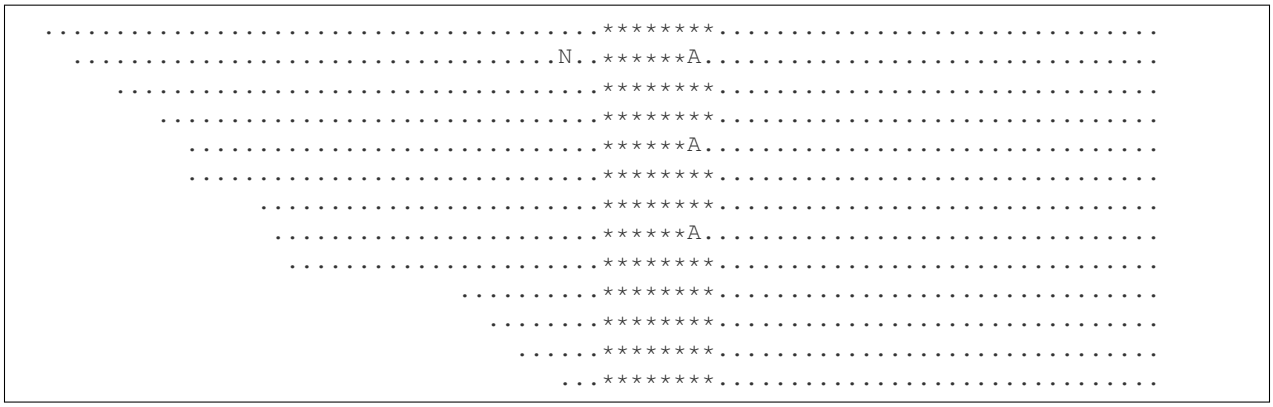
The parameter `bandwidth` controls the bandwidth of the banded alignment used in the alignment of reads against the profile. If `includeReference` is `true` then the reference is added as a pseudo-read (a new read at the end of the read store). This can be used for computing alignments of the reads agains the original reference.

```
#include <seqan/store.h>
#include <seqan/realign.h>
```

```
using namespace seqan;

int main()
{
    // Build paths to reference and to alignment.
    std::string refPath = getAbsolutePath("demos/tutorial/realignment/ref.fa");
    std::string samPath = getAbsolutePath("demos/tutorial/realignment/reads.sam");

    // Declare fragment store.
    FragmentStore<> store;

    // Load contigs and read alignment.
    loadContigs(store, refPath.c_str());
    BamFileIn bamFileIn(samPath.c_str());
    readRecords(store, bamFileIn);

    // Perform the realignment.
    reAlignment(store, /*contigID=*/ 0, /*method=*/ 1, /*bandwidth=*/ 20,
                /*includeReference=*/ true);

    // Layout alignment and print.
    AlignedReadLayout layout;
    layoutAlignment(layout, store);
    printAlignment(std::cout, layout, store, /*contigID=*/ 0, /*posBegin=*/ 1060,
                   /*posEnd=*/ 1140, /*lineBegin=*/ 0, /*lineEnd=*/ 100);

    return 0;
}
```

Here is the program's output. The reference pseudo-read is here shown as the first read (second row) below the reference (first row).

```
TTGACTGTGGGAGGATACATCTCTCCATCAATTATCTAAAA-------TAAATAAATAAACATCAGTTAAAAAGTTAAGG
.........................................AAATAAA................................
.........................................          .............................
.........................................******T.        .......................
.................................C.......********.....     .....................
.........................................******A...C...          ...............
.........................................********.........           ...........
......................................G..******A...........               ......
...................................G.....******A...G..........                ..
..............................G..........******A...............
.........................................******A................
.........................................******A.....................
.........................................********.............................
.......G.................................********..........C..................T.
.........................................******A...............................
.........................................******A................................
.........................................********.....................C.........
.........................................******A.............................C..
.........................................********...............................
.........................................******A................................
.........................................******A................................
.........................................********...............................
.........................................******A................................
.........................................********...............................
....................................C....******A................................
```



ToC

Contents

Seed Extension

Learning Objective You will learn how to do seed-and-extend with SeqAn, how to do local and global chaining of seeds. Finally, you will learn how to create a banded alignment around a seed chain.

Difficulty Average

Duration 2 h

Prerequisites *Sequences*, *Seeds*

Overview

Many efficient heuristics to find high scoring, but inexact, local alignments between two sequences start with small exact (or at least highly similar) segments, so called **seeds**, and extend or combine them to get larger highly similar regions. Probably the most prominent tool of this kind is BLAST *[AGM+90]*, but there are many other examples like FASTA *[Pea90]* or LAGAN *[BDC+03]*.

SeqAn's header file for all data structures and functions related to two-dimensional seeds is `<seqan/seeds.h>`.

Seed Extension

Seeds are often created quickly using a *k*-mer index: When a *k*-mer of a given length is found in both sequences, we can use it as a seed. However, the match can be longer than just *k* characters. To get longer matches, we use **seed extension**.

In the most simple case we simply look for matching characters in both sequences to the left and right end of the seed. This is called **match extension** and available through the extendSeed function using the `MatchExtend` tag. Below example shows how to extend seeds to the right end.

```
    // The horizontal and vertical sequence (subject and query sequences).
    CharString seqH = "The quick BROWN fox jumped again!";
    CharString seqV =     "thick BROWNIES for me!";
    // Create and print the seed sequence.
    Seed<Simple> seed(11, 7, 14, 10);
    std::cout << "original\n"
              << "seedH: " << infix(seqH, beginPositionH(seed),
                                    endPositionH(seed)) << "\n"
              << "seedV: " << infix(seqV, beginPositionV(seed),
                                    endPositionV(seed)) << "\n";

    // Perform match extension.
    extendSeed(seed, seqH, seqV, EXTEND_RIGHT, MatchExtend());
    // Print the resulting seed.
    std::cout << "result\n"
              << "seedH: " << infix(seqH, beginPositionH(seed),
                                    endPositionH(seed)) << "\n"
              << "seedV: " << infix(seqV, beginPositionV(seed),
                                    endPositionV(seed)) << "\n";
```

```
original
seedH: ROW
seedV: ROW
result
seedH: ROWN
seedV: ROWN
```

Assignment 1

Type Review

Objective Change the example from above to extend the seed to both sides.

Solution

```
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/seeds.h>

using namespace seqan;

int main()
{
    // The horizontal and vertical sequence (subject and query sequences).
    CharString seqH = "The quick BROWN fox jumped again!";
    CharString seqV =     "thick BROWNIES for me!";
    // Create and print the seed sequence.
```

```
    Seed<Simple> seed(11, 7, 14, 10);
    std::cout << "original\n"
              << "seedH: " << infix(seqH, beginPositionH(seed),
                                    endPositionH(seed)) << "\n"
              << "seedV: " << infix(seqV, beginPositionV(seed),
                                    endPositionV(seed)) << "\n";

    // Perform match extension.
    extendSeed(seed, seqH, seqV, EXTEND_BOTH, MatchExtend());
    // Print the resulting seed.
    std::cout << "result\n"
              << "seedH: " << infix(seqH, beginPositionH(seed),
                                    endPositionH(seed)) << "\n"
              << "seedV: " << infix(seqV, beginPositionV(seed),
                                    endPositionV(seed)) << "\n";

    return 0;
}
```

A more complex case is the standard bioinformatics approach of **x-drop extension**.

In the ungapped case, we extend the seed by comparing the i-th character to the left/right of the seed of the horizontal sequence (subject sequence) with the j-th character to the left/right of the seed in the vertical sequence (query sequence). Matches and mismatches are assigned with scores (usually matches are assigned with positive scores and mismatches are assigned with negative scores). The scores are summed up. When one or more mismatches occur, the running total will drop. When the sum drops more than a value x, the extension is stopped.

This approach is also available in the gapped case in the SeqAn library. Here, creating gaps is also possible but also assigned negative scores.

```
    // The horizontal and vertical sequence (subject and query sequences).
    CharString seqH = "The quick BROWN fox jumped again!";
    CharString seqV =     "thick BROWN boxes of brownies!";
    // Create and print the seed sequence.
    Seed<Simple> seed(11, 7, 14, 10);
    std::cout << "original\n"
              << "seedH: " << infix(seqH, beginPositionH(seed),
                                    endPositionH(seed)) << "\n"
              << "seedV: " << infix(seqV, beginPositionV(seed),
                                    endPositionV(seed)) << "\n";

    // Perform match extension.
    Score<int, Simple> scoringScheme(1, -1, -1);
    extendSeed(seed, seqH, seqV, EXTEND_BOTH, scoringScheme, 3,
               UnGappedXDrop());
    // Print the resulting seed.
    std::cout << "result\n"
              << "seedH: " << infix(seqH, beginPositionH(seed),
                                    endPositionH(seed)) << "\n"
              << "seedV: " << infix(seqV, beginPositionV(seed),
                                    endPositionV(seed)) << "\n";
```

```
original
seedH: ROW
seedV: ROW
result
seedH: ick BROWN fox
seedV: ick BROWN box
```

Assignment 2

Type Review

Objective Change the example from above to use gapped instead of ungapped x-drop extension.

Solution

```
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/score.h>
#include <seqan/seeds.h>

using namespace seqan;

int main()
{
    // The horizontal and vertical sequence (subject and query sequences).
    CharString seqH = "The quick BROWN fox jumped again!";
    CharString seqV =     "thick BROWN boxes of brownies!";
    // Create and print the seed sequence.
    Seed<Simple> seed(11, 7, 14, 10);
    std::cout << "original\n"
              << "seedH: " << infix(seqH, beginPositionH(seed),
                              endPositionH(seed)) << "\n"
              << "seedV: " << infix(seqV, beginPositionV(seed),
                              endPositionV(seed)) << "\n";

    // Perform match extension.
    Score<int, Simple> scoringScheme(1, -1, -1);
    extendSeed(seed, seqH, seqV, EXTEND_BOTH, scoringScheme, 3,
               GappedXDrop());
    // Print the resulting seed.
    std::cout << "result\n"
              << "seedH: " << infix(seqH, beginPositionH(seed),
                              endPositionH(seed)) << "\n"
              << "seedV: " << infix(seqV, beginPositionV(seed),
                              endPositionV(seed)) << "\n";

    return 0;
}
```

After extending a seed, we might wish to actually get the resulting alignment. When using gapped x-drop extension, we need to perform a banded global alignment of the two sequence infixes that correspond to the seed. This is shown in the following example:

```
    // The horizontal and vertical sequence (subject and query sequences).
    CharString seqH = "The quick BROWN fox jumped again!";
    CharString seqV =     "thick BROWN boxes of brownies!";
    // Create the seed sequence.
    Seed<Simple> seed(11, 7, 14, 10);

    // Perform match extension.
    Score<int, Simple> scoringScheme(1, -1, -1);
    extendSeed(seed, seqH, seqV, EXTEND_BOTH, scoringScheme, 3,
               GappedXDrop());

    // Perform a banded alignment.
    Align<CharString> align;
```

```
    resize(rows(align), 2);
    assignSource(row(align, 0), infix(seqH, beginPositionH(seed),
                                      endPositionH(seed)));
    assignSource(row(align, 1), infix(seqV, beginPositionV(seed),
                                      endPositionV(seed)));

    globalAlignment(align, scoringScheme);
    std::cout << "Resulting alignment\n" << align << "\n";
```

```
Resulting alignment
      0     .    :    .    :
         quick BROWN fox-- ju
           |||||||||| ||  |
        -thick BROWN boxes of
```

Assignment 3

Type Review

Objective Change the example from above to a gap open score of `-2` and a gap extension score of `-2`.

Solution Note that we do not have to explicitely call Gotoh's algorithm in `globalAlignment()`. The fact that the gap extension score is different from the gap opening score is enough.

```
#include <seqan/align.h>
#include <seqan/stream.h>
#include <seqan/score.h>
#include <seqan/seeds.h>
#include <seqan/sequence.h>

using namespace seqan;

int main()
{
    // The horizontal and vertical sequence (subject and query sequences).
    CharString seqH = "The quick BROWN fox jumped again!";
    CharString seqV =     "thick BROWN boxes of brownies!";
    // Create the seed sequence.
    Seed<Simple> seed(11, 7, 14, 10);

    // Perform match extension.
    Score<int, Simple> scoringScheme(1, -1, -2, -2);
    extendSeed(seed, seqH, seqV, EXTEND_BOTH, scoringScheme, 3,
               GappedXDrop());

    // Perform a banded alignment.
    Align<CharString> align;
    resize(rows(align), 2);
    assignSource(row(align, 0), infix(seqH, beginPositionH(seed),
                                      endPositionH(seed)));
    assignSource(row(align, 1), infix(seqV, beginPositionV(seed),
                                      endPositionV(seed)));

    globalAlignment(align, scoringScheme);
    std::cout << "Resulting alignment\n" << align << "\n";
```

```
    return 0;
}
```

Local Chaining using Seed Sets

Usually, we quickly determine a large number of seeds. When a seed is found, we want to find a "close" seed that we found previously and combine it to form a longer seed. This combination is called **local chaining**. This approach has been pioneered in the CHAOS and BLAT programs.

SeqAn provides the SeedSet class as a data structure to efficiently store seeds and combine new seeds with existing ones. The following example creates a SeedSet object `seeds`, adds four seeds to it and then prints its contents.

```
    typedef Seed<Simple>    TSeed;
    typedef SeedSet<TSeed> TSeedSet;

    TSeedSet seedSet;
    addSeed(seedSet, TSeed(0, 0, 2), Single());
    addSeed(seedSet, TSeed(3, 5, 2), Single());
    addSeed(seedSet, TSeed(4, 2, 3), Single());
    addSeed(seedSet, TSeed(9, 9, 2), Single());

    std::cout << "Resulting seeds.\n";
    typedef Iterator<TSeedSet>::Type TIter;
    for (TIter it = begin(seedSet, Standard()); it != end(seedSet, Standard()); ++it)
        std::cout << "(" << beginPositionH(*it) << ", " << endPositionH(*it)
                  << ", " << beginPositionV(*it) << ", " << endPositionV(*it)
                  << ", " << lowerDiagonal(*it) << ", " << upperDiagonal(*it)
                  << ")\n";
```

The output of the program above can be seen below.

```
Resulting seeds.
(3, 5, 5, 7, -2, -2)
(0, 2, 0, 2, 0, 0)
(9, 11, 9, 11, 0, 0)
(4, 7, 2, 5, 2, 2)
```

Note that we have used the `Single()` tag for adding the seeds. This forces the seeds to be added independent of the current seed set contents.

By using different overloads of the addSeed, we can use various local chaining strategies when adding seed `A`.

`Merge` If there is a seed `B` that overlaps with `A` and the difference in diagonals is smaller than a given threshold then `A` can be merged with `B`.

`SimpleChain` If there is a seed `B` whose distance in both sequences is smaller than a given threshold then `A` can be chained to `B`.

`Chaos` Following the strategy of CHAOS *[BCGottgens+03]*, if `A` is within a certain distance to `B` in both sequences and the distance in diagonals is smaller than a given threshold then `A` can be chained to `B`.

The addSeed function returns a boolean value indicating success in finding a suitable partner for chaining. A call using the `Single` strategy always yields `true`.

The following example shows how to use the `SimpleChain` strategy.

```
    typedef Seed<Simple>   TSeed;
    typedef SeedSet<TSeed> TSeedSet;
```

```
    Dna5String seqH;
    Dna5String seqV;
    Score<int, Simple> scoringScheme(1, -1, -1);

    String<TSeed> seeds;
    appendValue(seeds, TSeed(0, 0, 2));
    appendValue(seeds, TSeed(3, 5, 2));
    appendValue(seeds, TSeed(4, 2, 3));
    appendValue(seeds, TSeed(9, 9, 2));

    TSeedSet seedSet;
    for (unsigned i = 0; i < length(seeds); ++i)
    {
        if (!addSeed(seedSet, seeds[i], 2, 2, scoringScheme,
                     seqH, seqV, SimpleChain()))
            addSeed(seedSet, seeds[i], Single());
    }

    std::cout << "Resulting seeds.\n";
    typedef Iterator<TSeedSet>::Type TIter;
    for (TIter it = begin(seedSet, Standard());
         it != end(seedSet, Standard()); ++it)
        std::cout << "(" << beginPositionH(*it) << ", " << endPositionH(*it)
                  << ", " << beginPositionV(*it) << ", " << endPositionV(*it)
                  << ", " << lowerDiagonal(*it) << ", " << upperDiagonal(*it)
                  << ")\n";
```

As we can see, the seed `TSeed(4, 2, 3)` has been chained to `TSeed(0, 0, 2)`.

```
Resulting seeds.
(3, 5, 5, 7, -2, -2)
(0, 7, 0, 5, 0, 2)
(9, 11, 9, 11, 0, 0)
```

Assignment 4

Type Review

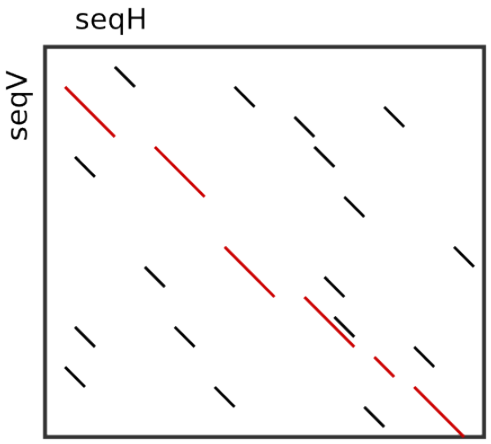
Objective Change the example above to use the `Chaos` strategy instead of the `SimpleChain`.

Solution

```
#include <seqan/stream.h>
#include <seqan/score.h>
#include <seqan/seeds.h>
#include <seqan/sequence.h>

using namespace seqan;

int main()
{
    typedef Seed<Simple>    TSeed;
    typedef SeedSet<TSeed> TSeedSet;

    Dna5String seqH;
    Dna5String seqV;
```

```
    Score<int, Simple> scoringScheme(1, -1, -1);

    String<TSeed> seeds;
    appendValue(seeds, TSeed(0, 0, 2));
    appendValue(seeds, TSeed(3, 5, 2));
    appendValue(seeds, TSeed(4, 2, 3));
    appendValue(seeds, TSeed(9, 9, 2));

    TSeedSet seedSet;
    for (unsigned i = 0; i < length(seeds); ++i)
    {
        if (!addSeed(seedSet, seeds[i], 2, 2, scoringScheme,
                     seqH, seqV, Chaos()))
            addSeed(seedSet, seeds[i], Single());
    }

    std::cout << "Resulting seeds.\n";
    typedef Iterator<TSeedSet>::Type TIter;
    for (TIter it = begin(seedSet, Standard());
         it != end(seedSet, Standard()); ++it)
        std::cout << "(" << beginPositionH(*it) << ", " << endPositionH(*it)
                  << ", " << beginPositionV(*it) << ", " << endPositionV(*it)
                  << ", " << lowerDiagonal(*it) << ", " << upperDiagonal(*it)
                  << ")\n";

    return 0;
}
```

Global Chaining



After one has determined a set of candidate seeds, a lot of these seeds will conflict. The image to the right shows an example. Some conflicting seeds might be spurious matches or come from duplication events.

Often, we need to find a linear ordering of the seeds such that each seed starts after all of its predecessor end in both sequences. This can be done efficiently using dynamic sparse programming (in time $\mathcal{O}(nlogn)$ where n is the number of seeds) as described in *[Gus97]*. The red seeds in the image to the right show such a valid chain.

This functionality is available in SeqAn using the chainSeedsGlobally function. The function gets a sequence container of Seed objects for the result as its first parameter and a SeedSet as its second parameter. A subset of the seeds from the SeedSet are then selected and stored in the result sequence.

The following shows a simple example.

```
typedef Seed<Simple>    TSeed;
typedef SeedSet<TSeed> TSeedSet;

TSeedSet seedSet;
addSeed(seedSet, TSeed(0, 0, 2), Single());
addSeed(seedSet, TSeed(3, 5, 2), Single());
addSeed(seedSet, TSeed(4, 2, 3), Single());
addSeed(seedSet, TSeed(9, 9, 2), Single());

String<TSeed> result;
chainSeedsGlobally(result, seedSet, SparseChaining());
```

Assignment 5

Type Review

Objective Change the example from above to use a different chain of seeds. The seeds should be `TSeed(1, 1, 3)`, `TSeed(6, 9, 2)`, `TSeed(10, 13, 3)`, and `TSeed(20, 22, 5)`.

Solution

```
#include <seqan/sequence.h>
#include <seqan/stream.h>
#include <seqan/seeds.h>

using namespace seqan;

int main()
{
    typedef Seed<Simple>    TSeed;
    typedef SeedSet<TSeed> TSeedSet;

    TSeedSet seedSet;
    addSeed(seedSet, TSeed(1, 1, 3), Single());
    addSeed(seedSet, TSeed(6, 9, 2), Single());
    addSeed(seedSet, TSeed(10, 13, 3), Single());
    addSeed(seedSet, TSeed(20, 22, 5), Single());

    String<TSeed> result;
    chainSeedsGlobally(result, seedSet, SparseChaining());

    return 0;
}
```

Banded Chain Alignment

After obtaining such a valid seed chain, we would like to obtain an alignment along the chain. For this, we can use the so-called banded chain alignment algorithm *[BDC+03]*. Around seeds, we can use banded DP alignment and the spaces between seeds can be aligned using standard DP programming alignment.

In SeqAn you can compute the banded chain alignment by calling the function bandedChainAlignment. This function gets the structure in which the alignment should be stored as the first parameter. This corresponds to the interface of the globalAlignment and allows the same input types. Additionally, this function requires a non-empty, non-decreasing

monotonic chain of seeds which is used as the rough global map for computing the global alignment. The third required parameter is the Score.

Note, that there are a number of optional parameters that can be specified. These include a second Score which, if specified, is used to evaluate the regions between two consecutive seeds differently than the regions around the seeds itself (for which then the first specified score is taken.). As for the global alignment you can use the AlignConfig to specify the behavior for initial and end gaps. The last optional parameter is the band extension. This parameter specifies to which size the bands around the anchors should be extended. The default value is 15 and conforms the default value in the LAGAN-algorithm *[BDC+03]*.

Important: At the moment the specified value for the band extension must be at least one.

```
    typedef Seed<Simple> TSeed;

    Dna5String sequenceH = "CGAATCCATCCCACACA";
    Dna5String sequenceV = "GGCGATNNNCATGGCACA";

    String<TSeed> seedChain;
    appendValue(seedChain, TSeed(0, 2, 5, 6));
    appendValue(seedChain, TSeed(6, 9, 9, 12));
    appendValue(seedChain, TSeed(11, 14, 17, 16));

    Align<Dna5String, ArrayGaps> alignment;
    resize(rows(alignment), 2);
    assignSource(row(alignment, 0), sequenceH);
    assignSource(row(alignment, 1), sequenceV);

    Score<int, Simple> scoringScheme(2, -1, -2);

    int result = bandedChainAlignment(alignment, seedChain, scoringScheme, 2);

    std::cout << "Score: " << result << std::endl;
    std::cout << alignment << std::endl;
```

The output of the example above.

```
Score: 5
      0     .    :    .    :
        --CGAAT--CCATCCCACACA
          || ||   |||    ||||
        GGCG-ATNNNCATGG--CACA
```

Assignment 6

Type Review

Objective Change the example form above to use two different scoring schemes. The scoring scheme for the seeds should use the Levenshtein distance and the score for the gap regions should be an affine score with the following values: match = 2, mismatch = -1, gap open = -2, gap extend = -1.

Furthermore, we are looking for a semi-global alignment here the initial and end gaps in the query sequence are free.

Solution

```
#include <seqan/sequence.h>
#include <seqan/align.h>
#include <seqan/score.h>
#include <seqan/seeds.h>

using namespace seqan;

int main()
{
    typedef Seed<Simple>    TSeed;

    Dna5String sequenceH = "CGAATCCATCCCACACA";
    Dna5String sequenceV = "GGCGATNNNCATGGCACA";
    Score<int, Simple> scoringSchemeAnchor(0, -1, -1);
    Score<int, Simple> scoringSchemeGap(2, -1, -1, -2);

    String<TSeed> seedChain;
    appendValue(seedChain, TSeed(0, 2, 5, 6));
    appendValue(seedChain, TSeed(6, 9, 9, 12));
    appendValue(seedChain, TSeed(11, 14, 17, 16));

    Align<Dna5String, ArrayGaps> alignment;
    resize(rows(alignment), 2);
    assignSource(row(alignment, 0), sequenceH);
    assignSource(row(alignment, 1), sequenceV);
    AlignConfig<true, false, false, true> alignConfig;

    int result = bandedChainAlignment(alignment, seedChain, scoringSchemeAnchor,␣
↪scoringSchemeGap, alignConfig, 2);

    std::cout << "Score: " << result << std::endl;
    std::cout << alignment << std::endl;

    return 0;
}
```

ToC

Contents

Graph Algorithms

Learning Objective This tutorial shows how to use some graph algorithms in SeqAn. In particular we will use the dijkstra algorithm to find shortest path and viterbi Algorithm to compute Viterbi path of a sequence.

Difficulty Average

Duration 1 h

Prerequisites *Graphs*

Overview

The following graph algorithms are currently available in SeqAn:

Elementary Graph Algorithms

- Breadth-First Search (breadthFirstSearch)
- Depth-First Search (depthFirstSearch)
- Topological Sort (topologicalSort)
- Strongly Connected Components (stronglyConnectedComponents)

Minimum Spanning Tree

- Prim's Algorithm (primsAlgorithm)
- Kruskal's Algorithm (kruskalsAlgorithm)

Single-Source Shortest Path

- DAG Shortest Path (dagShortestPath)
- Bellman-Ford (bellmanFordAlgorithm)
- Dijkstra (dijkstra)

All-Pairs Shortest Path

- All-Pairs Shortest Path (allPairsShortestPath)
- Floyd Warshall (floydWarshallAlgorithm)

Maximum Flow

- Ford-Fulkerson (fordFulkersonAlgorithm)

Transitive Closure

- Transitive Closure (transitiveClosure)

Bioinformatics Algorithms

- Needleman-Wunsch (globalAlignment)
- Gotoh (globalAlignment)
- Hirschberg with Gotoh (globalAlignment)
- Smith-Waterman (localAlignment)
- Multiple Sequence Alignment (globalMsaAlignment)
- UPGMA (upgmaTree)
- Neighbor Joining (njTree)

The biological algorithms use heavily the alignment graph. Most of them are covered in the tutorial *Alignment*. All others use the appropriate standard graph. All algorithms require some kind of additional input, e.g., the Dijkstra algorithm requires a distance property map, alignment algorithms sequences and a score type and the network flow algorithm capacities on the edges.

Generally, only a single function call is sufficient to carry out all the calculations of a graph algorithm. In most cases you will have to define containers that store the algorithms results prior to the function call.

In our example, we apply the shortest-path algorithm of Dijkstra. It is implemented in the function dijkstra.

Let's have a look at the input parameters. The first parameter is of course the graph, `g`. Second, you will have to specify a vertex descriptor. The function will compute the distance from this vertex to all vertices in the graph. The last input parameter is an edge map containing the distances between the vertices. One may think that the distance map is already contained in the graph. Indeed this is the case for our graph type but it is not in general. The cargo of a graph might as well be a string of characters or any other type. So, we first have to find out how to access our internal edge map. We do not need to copy the information to a new map. Instead we can define an object of the type InternalPropertyMap of our type `TCargo`. It will automatically find the edge labels in the graph when the function property or getProperty is called on it with the corresponding edge descriptor.

The output containers of the shortest-path algorithm are two property maps, `predMap` and `distMap`. The `predMap` is a vertex map that determines a shortest-paths-tree by mapping the predecessor to each vertex. Even though we are not interested in this information, we have to define it and pass it to the function. The `distMap` indicates the length of the shortest path to each vertex.

```
    typedef Size<TGraph>::Type TSize;
    InternalPropertyMap<TCargo> cargoMap;
    String<TVertexDescriptor> predMap;
    String<TSize> distMap;
```

Having defined all these property maps, we can then call the function dijkstra:

```
    dijkstra(predMap, distMap, g, vertHannover, cargoMap);
```

Finally, we have to output the result. Therefore, we define a second vertex iterator `itV2` and access the distances just like the city names with the function property on the corresponding property map.

```
    TVertexIterator itV2(g);
    while (!atEnd(itV2))
    {
        std::cout << "Shortest path from " << property(cityNames, vertHannover) << "␣
↪to " << property(cityNames, value(itV2)) << ": ";
        std::cout << property(distMap, value(itV2)) << std::endl;
        goNext(itV2);
    }

    return 0;
}
```

Assignment 1

Type Application

Objective Write a program which calculates the connected components of the graph defined in *Assignment 2* of the Graphs tutorial and Output the connected component for each vertex.

Solution SeqAn provides the function stronglyConnectedComponents to compute the connected components of a directed graph. The first parameter of this function is of course the graph. The second parameter is an output parameter. It is a vertex map that will map a component id to each vertex. Vertices that share the same id are in the same component.

```
    String<unsigned int> component;
    stronglyConnectedComponents(component, g);
```

Now, the only thing left to do is to walk through our graph and ouput each vertex and the corresponding component using the function getProperty. One way of doing so is to define a VertexIterator.

```
    std::cout << "Strongly Connected Components: " << std::endl;
    typedef Iterator<TGraph, VertexIterator>::Type TVertexIterator;
    TVertexIterator it(g);
    while (!atEnd(it))
    {
        std::cout << "Vertex " << getProperty(nameMap, getValue(it)) << ": ";
        std::cout << "Component = " << getProperty(component, getValue(it)) <<␣
↪std::endl;
        goNext(it);
    }
```

```
    return 0;
}
```

The output for the graph defined in the *Assignment 1* looks as follows:

```
Strongly Connected Components:
Vertex a: Component = 3
Vertex b: Component = 3
Vertex c: Component = 2
Vertex d: Component = 2
Vertex e: Component = 3
Vertex f: Component = 1
Vertex g: Component = 1
Vertex h: Component = 0
```

The graph consists of four components. The first contains vertex `a`, `b`, and `e`, the second contains vertex `c` and `d`, the third contains vertex `f` and `g` and the last contains only vertex `h`.

Assignment 2

Type Application

Objective Extend the program from the *Assignment 3* of the Graphs tutorial. Given the sequence `s = "CTTCATGTGAAAGCAGACGTAAGTCA"`.

1. calculate the Viterbi path of `s` and output the path as well as the probability of the path and
2. calculate the probability that the HMM generated `s` with the forward and backward algorithm.

Solution In *Assignment 3* of the Graphs tutorial we defined an HMM with three states: exon, splice, and intron.

The Viterbi path is the sequence of states that is most likely to produce a given output. In SeqAn, it can be calculated with the function viterbiAlgorithm. The produced output for this assignment is the DNA sequence `s`.

The first parameter of the function viterbiAlgorithm is of course the HMM, and the second parameter is the sequence `s`. The third parameter is an output parameter that will be filled by the function. Since we want to compute a sequence of states, this third parameter is a String of VertexDescriptors which assigns a state to each character of the sequence `s`.

The return value of the function viterbiAlgorithm is the overall probability of this sequence of states, the Viterbi path.

The only thing left is to output the path. The path is usually longer than the given sequence. This is because the HMM may have silent states, e.g. the begin and end state. To check if a state is silent SeqAn provides the function isSilent.

```
    String<Dna> sequence = "CTTCATGTGAAAGCAGACGTAAGTCA";
    String<TVertexDescriptor> path;
    TProbability p = viterbiAlgorithm(path, hmm, sequence);
    std::cout << "Viterbi algorithm" << std::endl;
    std::cout << "Probability of best path: " << p << std::endl;
    std::cout << "Sequence: " << std::endl;
    for (TSize i = 0; i < length(sequence); ++i)
        std::cout << sequence[i] << ',';
    std::cout << std::endl;
    std::cout << "State path: " << std::endl;
    for (TSize i = 0; i < length(path); ++i)
    {
        std::cout << path[i];
        if (isSilent(hmm, path[i]))
            std::cout << " (Silent)";
        if (i < length(path) - 1)
            std::cout << ',';
    }
    std::cout << std::endl;
```

The output of the above piece of code is:

```
Viterbi algorithm
Probability of best path: 1.25465e-18
Sequence:
C,T,T,C,A,T,G,T,G,A,A,A,G,C,A,G,A,C,G,T,A,A,G,T,C,A,
State path:
0 (Silent),1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,3,3,3,3,3,3,3,4 (Silent)
```

It is even simpler to use the forward algorithm in SeqAn since it needs only the HMM and the sequence as parameters and returns a single probability. This is the probability of the HMM to generate the given sequence. The corresponding function is named forwardAlgorithm.

```
    std::cout << "Forward algorithm" << std::endl;
    p = forwardAlgorithm(hmm, sequence);
    std::cout << "Probability that the HMM generated the sequence: " << p <<␣
↪std::endl;
```

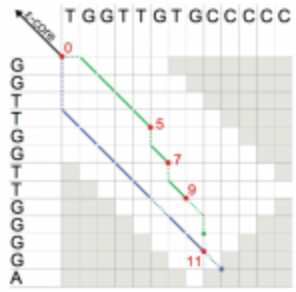
Analogously, the function backwardAlgorithm implements the backward algorithm in SeqAn.

```
    std::cout << "Backward algorithm" << std::endl;
    p = backwardAlgorithm(hmm, sequence);
    std::cout << "Probability that the HMM generated the sequence: " << p <<␣
↪std::endl;
    return 0;
}
```

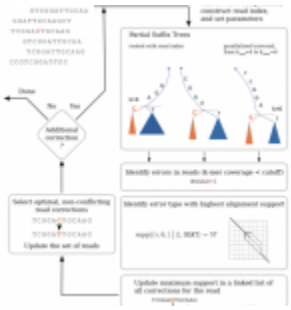
The output of these two code fragments is:

```
Forward algorithm
Probability that the HMM generated the sequence: 2.71585e-18
Backward algorithm
Probability that the HMM generated the sequence: 2.71585e-18
```

SeqAn contains many efficient implementations of core bioinformatics algorithms. This starts with the standard dynamic programming based alignment algorithms with all its subtypes. Global alignment, local alignment, banded and unbanded, for proteins or DNA, using seeds or not, needing a traceback or not. Check out nearly 200 combinations of this module which incidentally will soon be fully multithreaded and SIMD accelerated.



A depiction of the extension phase of the Stellar algorithm.



A depiction of the error correcting algorithm in the Fiona algorithm.

SeqAn contains algorithms for read mapping based on q-gram or string indices, multiple alignment algorithms, filter algorithms for string search as well es error correction methods.

The algorithms are usually generic in the sense that they can be configured via template arguments and usually work for many, if not arbitrary alphabets. SeqAn applications are usually short, very maintainable combinations of those core algorithmic components. Being well defined, the SeqAn components are quite amenable to optimisation and acceleration using multicore computing, vectorisation or accelerators.

On the right you will find several tutorials about SeqAn's algorithms. Under *Pattern Matching* you will find tutorials for online or indexed pattern search. Under *DP Alignment* you will find tutorials for all versions of DP based alignments (pairwise and multiple) and as special cases consensus alignment under *Consensus Alignment* and realignment algorithms under *Realignment*.

Under *Seed Extension* you will find tutorials for SeqAn's seed module and various extension algorithms. And finally you can find under *Graph Algorithms* algorithms that work on SeqAn's graph type.

Input/Output

ToC

Contents

File I/O Overview

Learning Objective This article will give you an overview of the formatted file I/O in SeqAn.

Difficulty Basic

Duration 30 min

Prerequisites *Sequences*

Overview

Most file formats in bioinformatics are structured as lists of records. Often, they start out with a header that itself contains different header records. For example, the Binary Sequence Alignment/Map (SAM/BAM) format starts with an header that lists all contigs of the reference sequence. The BAM header is followed by a list of BAM alignment records that contain query sequences aligned to some reference contig.

Formatted Files

SeqAn allows to read or write record-structured files through two types of classes: FormattedFileIn and FormattedFileOut. Classes of type FormattedFileIn allow to read files, whereas classes of type FormattedFileOut allow to write files. Note how these types of classes **do not allow to read and write the same file at the same time**.

These types of classes provide the following I/O operations on formatted files:

1. Open a file given its filename or attach to an existing stream like *std::cin* or *std::cout*.
2. Guess the file format from the file content or filename extension.
3. Access compressed or uncompressed files transparently.

SeqAn provides the following file formats:

- SeqFileIn, SeqFileOut (see Tutorial *Sequence I/O*)
- BamFileIn, BamFileOut (see Tutorial *SAM and BAM I/O*)

- BedFileIn, BedFileOut (see Tutorial *BED I/O*)
- VcfFileIn, VcfFileOut (see Tutorial *VCF I/O*)
- GffFileIn, GffFileOut (see Tutorial *GFF and GTF I/O*)
- RoiFileIn, RoiFileOut
- SimpleIntervalsFileIn, SimpleIntervalsFileInOut
- UcscFileIn, UcscFileOut

Warning: Access to compressed files relies on external libraries. For instance, you need to have zlib installed for reading `.gz` files and libbz2 for reading `.bz2` files. If you are using Linux or OS X and you followed the *Getting Started* tutorial closely, then you should have already installed the necessary libraries. On Windows, you will need to follow *Installing Dependencies* to get the necessary libraries.

You can check whether you have installed these libraries by running CMake again. Simply call `cmake .` in your build directory. At the end of the output, there will be a section "SeqAn Features". If you can read `ZLIB - FOUND` and `BZIP2 - FOUND` then you can use zlib and libbz2 in your programs.

Basic I/O

This tutorial shows the basic functionalities provided by any class of type FormattedFileIn or FormattedFileOut. In particular, this tutorial adopts the classes BamFileIn and BamFileOut as concrete types. The class BamFileIn allows to read files in SAM or BAM format, whereas the class BamFileOut allows to write them. Nonetheless, **these functionalities are independent from the particular file format** and thus valid for all record-based file formats supported by SeqAn.

The demo application shown here is a simple BAM to SAM converter.

Includes

Support for a specific format comes by including a specific header file. In this case, we include the BAM header file:

```
#include <seqan/bam_io.h>

using namespace seqan;

int main()
{
```

Opening and Closing Files

Classes of type FormattedFileIn and FormattedFileOut allow to open and close files.

A file can be opened by passing the filename to the constructor:

```
    CharString bamFileInName = getAbsolutePath("demos/tutorial/file_io_overview/
↪example.bam");
    CharString samFileOutName = getAbsolutePath("demos/tutorial/file_io_overview/
↪example.sam");

    // Open input BAM file, BamFileIn supports both SAM and BAM files.
    BamFileIn bamFileIn(toCString(bamFileInName));
```

```
    // Open output SAM file by passing the context of bamFileIn and the filename to␣
↪open.
    BamFileOut samFileOut(context(bamFileIn), toCString(samFileOutName));
```

Alternatively, a file can be opened after construction by calling open:

```
    // Alternative way to open a bam or sam file
    BamFileIn openBamFileIn;
    open(openBamFileIn, toCString(bamFileInName));
```

Note that any file is closed *automatically* whenever the FormattedFileIn or FormattedFileOut object goes out of scope. Eventually, a file can be closed *manually* by calling close.

Accessing the Header

To access the header, we need an object representing the format-specific header. In this case, we use an object of type BamHeader. The content of this object can be ignored for now, it will be covered in the *SAM and BAM I/O* tutorial.

```
    // Copy header.
    BamHeader header;
    readHeader(header, bamFileIn);
    writeHeader(samFileOut, header);
```

The function readHeader reads the header from the input BAM file and writeHeader writes it to the SAM output file.

Accessing the Records

Again, to access records, we need an object representing format-specific information. In this case, we use an object of type BamAlignmentRecord. Each call to readRecord reads one record from the BAM input file and moves the BamFileIn forward. Each call to writeRecord writes the record just read to the SAM output files. We check the end of the input file by calling atEnd.

```
    // Copy all records.
    BamAlignmentRecord record;
    while (!atEnd(bamFileIn))
    {
        readRecord(record, bamFileIn);
        writeRecord(samFileOut, record);
    }

    return 0;
}
```

Our small BAM to SAM conversion demo is ready. The tool still lacks error handling, reading from standard input and writing to standard output. You are now going to add these features.

Error Handling

We distinguish between two types of errors: *low-level* file I/O errors and *high-level* file format errors. Possible file I/O errors can affect both input and output files. Example of errors are: the file permissions forbid a certain operation, the file does not exist, there is a disk reading error, a file being read gets deleted while we are reading from it, or there is

a physical error in the hard disk. Conversely, file format errors can only affect input files: such errors arise whenever the content of the input file is incorrect or damaged. Error handling in SeqAn is implemented by means of exceptions.

I/O Errors

All FormattedFileIn and FormattedFileOut constructors and functions throw exceptions of type IOError to signal *low-level* file I/O errors. Therefore, it is sufficient to catch these exceptions to handle I/O errors properly.

There is only one exception to this rule. Function open returns a `bool` to indicate whether the file was opened successfully or not.

Assignment 1

Type Application

Objective Improve the program above to detect file I/O errors.

Hint Use the IOError class.

Solution

```
#include <seqan/bam_io.h>

using namespace seqan;

int main(int argc, char const ** argv)
{
    CharString bamFileInName = getAbsolutePath("demos/tutorial/file_io_overview/
↪example.bam");
    CharString samFileOutName = getAbsolutePath("demos/tutorial/file_io_overview/
↪example.sam");

    // Open input BAM file.
    BamFileIn bamFileIn;
    BamHeader header;
    if (!open(bamFileIn, toCString(bamFileInName)))
    {
        std::cerr << "ERROR: could not open input file " << bamFileInName << ".\n
↪";
        return 1;
    }

    // Open output SAM file.
    BamFileOut samFileOut(context(bamFileIn), toCString(samFileOutName));

    // Copy header.
    try
    {
        readHeader(header, bamFileIn);
        writeHeader(samFileOut, header);
    }
    catch (IOError const & e)
    {
        std::cerr << "ERROR: could not copy header. " << e.what() << "\n";
    }

    // Copy all records.
```

```
    BamAlignmentRecord record;
    while (!atEnd(bamFileIn))
    {
        try
        {
            readRecord(record, bamFileIn);
            writeRecord(samFileOut, record);
        }
        catch (IOError const & e)
        {
            std::cerr << "ERROR: could not copy record. " << e.what() << "\n";
        }
    }

    return 0;
}
```

Format Errors

Classes of types FormattedFileIn throw exceptions of type ParseError to signal *high-level* input file format errors.

Assignment 2

Type Application

Objective Improve the program above to detect file format errors.

Solution

```
#include <seqan/bam_io.h>

using namespace seqan;

int main(int argc, char const ** argv)
{
    CharString bamFileInName = getAbsolutePath("demos/tutorial/file_io_overview/
↪example.bam");
    CharString samFileOutName = getAbsolutePath("demos/tutorial/file_io_overview/
↪example.sam");

    // Open input BAM file.
    BamFileIn bamFileIn;
    if (!open(bamFileIn, toCString(bamFileInName)))
    {
        std::cerr << "ERROR: could not open input file " << bamFileInName << ".\n
↪";
        return 1;
    }

    // Open output SAM file.
    BamFileOut samFileOut(context(bamFileIn), toCString(samFileOutName));
    // Copy header.
    BamHeader header;
    try
    {
```

```
        readHeader(header, bamFileIn);
        writeHeader(samFileOut, header);
    }
    catch (ParseError const & e)
    {
        std::cerr << "ERROR: input header is badly formatted. " << e.what() << "\n
↪";
    }
    catch (IOError const & e)
    {
        std::cerr << "ERROR: could not copy header. " << e.what() << "\n";
    }

    // Copy all records.
    BamAlignmentRecord record;
    while (!atEnd(bamFileIn))
    {
        try
        {
            readRecord(record, bamFileIn);
            writeRecord(samFileOut, record);
        }
        catch (ParseError const & e)
        {
            std::cerr << "ERROR: input record is badly formatted. " << e.what() <
↪< "\n";
        }
        catch (IOError const & e)
        {
            std::cerr << "ERROR: could not copy record. " << e.what() << "\n";
        }
    }

    return 0;
}
```

Streams

The FormattedFileIn and FormattedFileOut constructors accept not only filenames, but also standard C++ streams, or any other class implementing the Stream concept. For instance, you can pass *std::cin* to any FormattedFileIn constructor and *std::cout* to any FormattedFileOut constructor.

Note: When writing to *std::cout*, classes of type FormattedFileOut cannot guess the file format from the filename extension. Therefore, the file format has to be specified explicitly by providing a tag, e.g. Sam or Bam.

Assignment 3

Type Application

Objective Improve the program above to write to standard output.

Solution

```
#include <seqan/bam_io.h>

using namespace seqan;

int main(int argc, char const ** argv)
{
    CharString bamFileInName = getAbsolutePath("demos/tutorial/file_io_overview/
↪example.bam");

    // Open input BAM file.
    BamFileIn bamFileIn;
    if (!open(bamFileIn, toCString(bamFileInName)))
    {
        std::cerr << "ERROR: could not open input file " << bamFileInName << ".\n
↪";
        return 1;
    }

    // Open output SAM which is the standard output.
    BamFileOut samFileOut(context(bamFileIn), std::cout, Sam());

    // Copy header.
    BamHeader header;
    try
    {
        readHeader(header, bamFileIn);
        writeHeader(samFileOut, header);
    }
    catch (ParseError const & e)
    {
        std::cerr << "ERROR: input header is badly formatted. " << e.what() << "\n
↪";
    }
    catch (IOError const & e)
    {
        std::cerr << "ERROR: could not copy header. " << e.what() << "\n";
    }

    // Copy all records.
    BamAlignmentRecord record;
    while (!atEnd(bamFileIn))
    {
        try
        {
            readRecord(record, bamFileIn);
            writeRecord(samFileOut, record);
        }
        catch (ParseError const & e)
        {
            std::cerr << "ERROR: input record is badly formatted. " << e.what() <
↪< "\n";
        }
        catch (IOError const & e)
        {
            std::cerr << "ERROR: could not copy record. " << e.what() << "\n";
        }
    }
```

```
    return 0;
}
```

Running this program results in the following output.

```
@HD         VN:1.3         SO:coordinate
@SQ         SN:ref         LN:45
@SQ         SN:ref2         LN:40
r001         163         ref         7         30         8M4I4M1D3M         =         37         39
↪                              XX:B:S,12561,2,20,112
r002         0         ref         9         30         1S2I6M1P1I1P1I4M2I         *         0         0
↪
r003         0         ref         9         30         5H6M         *         0         0         AGCTAA
↪
r004         0         ref         16         30         6M14N1I5M         *         0         0
↪
r003         16         ref         29         30         6H5M         *         0         0         TAGG
↪
r001         83         ref         37         30         9M         =         7         -
↪39         CAGCGCCAT
```

Next Steps

If you want, you can now have a look at the API documentation of the FormattedFile class.

You can now read the tutorials for **already supported file formats**:

- *Sequence I/O*
- *SAM and BAM I/O*
- *VCF I/O*
- *BED I/O*
- *GFF and GTF I/O*

ToC

Contents

Sequence I/O

Learning Objective You will learn how to read and write sequence files in FASTA, FASTQ, EMBL or GenBank format.

Difficulty Basic

Duration 20 min

Prerequisites *Sequences*, *File I/O Overview*

This tutorial explains how to read and write sequence files using the SeqFileIn and SeqFileOut classes. These classes provide an API for accessing sequence files in different file formats, either compressed or uncompressed.

FASTA/FASTQ Format

FASTA/FASTQ are record-based files. A FASTA record contains the sequence id and the sequence characters. Here is an example of FASTA file:

```
>seq1
CCCCCCCCCCCCCCC
>seq2
CGATCGATC
>seq3
TTTTTTT
```

In addition to that, a FASTQ record contains also a quality value for each sequence character. Here is an example of FASTQ file:

```
@seq1
CCCCCCCCCCCCCCC
+
IIIIIHIIIIIIIII
@seq2
CGATCGATC
+
IIIIIIIII
@seq3
TTTTTTT
+
IIIIHHG
```

SeqFile Formats

We can read sequence files with the SeqFileIn class and write them with the SeqFileOut class. These classes support files in FASTA, FASTQ, EMBL or GenBank format.

Note that SeqFileOut will guess the format from the file name.

File Format	File Extension
FASTA	`.fa`, `.fasta`
FASTQ	`.fq`, `.fastq`
EMBL	`.embl`
GenBank	`.gbk`

A First Working Example

Let us start out with a minimal working example. The following program reads a FASTA file called `example.fa` and prints out the identifier and the sequence of the first record.

```
#include <seqan/seq_io.h>

using namespace seqan;

int main()
{
    CharString seqFileName = getAbsolutePath("demos/tutorial/sequence_io/example.fa");
    CharString id;
    Dna5String seq;

    SeqFileIn seqFileIn(toCString(seqFileName));
    readRecord(id, seq, seqFileIn);
    std::cout << id << '\t' << seq << '\n';

    return 0;
}
```

We call the SeqFileIn constructor with the path to the file to read. Successively, we call the function readRecord to read the first record from the file. Note that, differently from all others FormattedFileIn classes, readRecord accepts **separate** identifier and sequence Strings rather than one single record object.

Assignment 1

Type Reproduction

Objective Copy the above example of a FASTA file in a new file `example.fa` in a directory of your choice.

> Copy the program above into a new application `basic_seq_io_example`, adjust the path `"example.fa"` to the just created FASTA file, compile the program, and run it.
>
> You should see the following output:
>
> ```
> seq1        CCCCCCCCCCCCCCC
> ```

Solution

> ```
> #include <seqan/seq_io.h>
>
> using namespace seqan;
>
> int main()
> {
>     CharString seqFileName = getAbsolutePath("demos/tutorial/sequence_io/example.
> ↪fa");
>     CharString id;
>     Dna5String seq;
>
>     SeqFileIn seqFileIn(toCString(seqFileName));
>     readRecord(id, seq, seqFileIn);
>     std::cout << id << '\t' << seq << '\n';
>
>     return 0;
> }
> ```

Handling Errors

As explained in the *File I/O Overview* tutorial, SeqFileIn and SeqFileOut throw exceptions to signal eventual errors. Invalid characters inside an input file will be signaled by readRecord via parsing exceptions.

Assignment 2

Type Application

Objective Improve the above program to handle errors.

Hint You can use the generic class Exception to catch both low-level and high-level I/O errors.

Solution

```
#include <seqan/seq_io.h>

using namespace seqan;

int main()
{
    CharString seqFileName = getAbsolutePath("demos/tutorial/sequence_io/example.
↪fa");
    CharString id;
    Dna5String seq;

    SeqFileIn seqFileIn;
    if (!open(seqFileIn, toCString(seqFileName)))
    {
        std::cerr << "ERROR: Could not open the file.\n";
        return 1;
    }

    try
    {
        readRecord(id, seq, seqFileIn);
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    std::cout << id << '\t' << seq << '\n';

    return 0;
}
```

Accessing Records in Batches

There are three use cases for reading or writing record-based files:

1. read or write the file **record by record**;

2. read or write a **batch of records**, e.g. 100k records at a time;
3. read or write **all records** from or to the file.

The class SeqFileIn provides the functions readRecord and readRecords, while the class SeqFileOut provides the functions writeRecord and writeRecords.

Tip: Reading records in batches is more efficient than reading single records.

Note that the function readRecords uses StringSet instead of String. By default, readRecords reads **all** remaining records. Optionally, one can specify a batch of records to be read.

```
    CharString seqFileName = getAbsolutePath("/demos/tutorial/sequence_io/example.fa
↪");

    StringSet<CharString> ids;
    StringSet<Dna5String> seqs;

    SeqFileIn seqFileIn(toCString(seqFileName));

    // Reads up to 10 records.
    readRecords(ids, seqs, seqFileIn, 10);

    // Reads all remaining records.
    readRecords(ids, seqs, seqFileIn);
```

Assignment 3

Type Application

Objective Change your program from above to load all sequences and print them in the same fashion.

You should be able to run your program on the example file we created above and see the following output:

```
seq1        CCCCCCCCCCCCCCC
seq2        CGATCGATC
seq3        TTTTTTT
```

Hint You can use the function readRecords to load all records at once.

Solution

```
#include <seqan/seq_io.h>

using namespace seqan;

int main()
{
    CharString seqFileName = getAbsolutePath("demos/tutorial/sequence_io/example.
↪fa");

    SeqFileIn seqFileIn;
    if (!open(seqFileIn, toCString(seqFileName)))
    {
        std::cerr << "ERROR: Could not open the file.\n";
        return 1;
    }
```

```
    StringSet<CharString> ids;
    StringSet<Dna5String> seqs;

    try
    {
        readRecords(ids, seqs, seqFileIn);
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    for (unsigned i = 0; i < length(ids); ++i)
        std::cout << ids[i] << '\t' << seqs[i] << '\n';

    return 0;
}
```

Accessing Qualities

Functions readRecord, readRecords, writeRecord and writeRecords are available in two variants:

1. the first variant accepts only the sequence identifier and sequence characters, besides the SeqFileIn object;
2. the second variant accepts an additional CharString for a PHRED base quality string.

If the first variant is used on an output file containing qualities, e.g. a FASTQ file, then writeRecord writes qualities as `'I'`, i.e. PHRED score 40. If the second variant is used on an input file containing no qualities, e.g. a FASTA file, then readRecord returns **empty** quality strings.

Here is an example for the second variant of readRecord:

```
    CharString seqFqFileName = getAbsolutePath("/demos/tutorial/sequence_io/example.fq
↪");
    CharString id;
    Dna5String seq;
    CharString qual;

    SeqFileIn seqFqFileIn(toCString(seqFqFileName));

    readRecord(id, seq, qual, seqFqFileIn);
```

Tip: When DnaQ or Dna5Q Strings are used, then you should use the second variant. The qualities are simply stored directly in the sequence characters.

Assignment 4

Type Application

Objective Copy the above example of FASTQ file in a new file `example.fq` in a directory of your choice.

Change your result of Assignment 3 to use the variant of readRecord that also reads in the qualities and writes them next to the sequences.

When your program is called on this file, the result should look as follows.

```
seq1        CCCCCCCCCCCCCCC
+qual:        IIIIIHIIIIIIII
seq2        CGATCGATC
+qual:        IIIIIIIII
seq3        TTTTTTT
+qual:        IIIIHHG
```

Solution

```
#include <seqan/seq_io.h>

using namespace seqan;

int main()
{
    CharString seqFileName = getAbsolutePath("demos/tutorial/sequence_io/example.
↪fq");

    SeqFileIn seqFileIn;
    if (!open(seqFileIn, toCString(seqFileName)))
    {
        std::cerr << "ERROR: Could not open the file.\n";
        return 1;
    }

    StringSet<CharString> ids;
    StringSet<Dna5String> seqs;
    StringSet<CharString> quals;

    try
    {
        readRecords(ids, seqs, quals, seqFileIn);
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    for (unsigned i = 0; i < length(ids); ++i)
        std::cout << ids[i] << '\t' << seqs[i] << "\n+qual:\t" << quals[i] << '\n
↪';

    return 0;
}
```

Next Steps

- Read the Wikipedia articles about the FASTA file format and the FASTQ file format and quality values to refresh your knowledge.
- Read the *Indexed FASTA I/O* tutorial to learn how to read FASTA files efficiently in a random-access fashion.
- Continue with the *Tutorials*.

ToC

Contents

Indexed FASTA I/O

Learning Objective :: In this tutorial, you will learn how to use a FASTA Index file (`.fai`) for indexed random-access to FASTA files. This is useful for retrieving regions (e.g. `chr1:123-10004`) or single sequences (e.g. `chr1`) from FASTA files quickly.

Difficulty Average

Duration 30 min

Prerequisites *Sequences*

The idea of FASTA index files (*FAI*) comes from the samtools program by Heng Li. The program provides a command `samtools faidx` for rapidly accessing parts of a large FASTA file (e.g. querying for the first chromosome by the identifier "chr1" or querying for 900 characters starting from character 100 (1-based) by `chr1:100-1,000`). To do this, the program creates an index file that contains one entry for each sequence. If the FASTA file is named `path/sequence.fasta`, the index file is usually named `path/sequence.fasta.fai`.

Using such index files, it is possible to rapidly read parts of the given sequence file. The module `<seqan/seq_io.h>` allows to create and read such `.fai` index files and exposes an API to read parts of FASTA file randomly.

Note: FASTA/FASTQ Meta Data and Sequence Ids

FASTA and FASTQ files have one metadata record for each sequence. This usually contains the sequence name but sometimes a lot of additional information is stored. There is no consensus for the metadata.

However, it is common to store the sequence identifier (*id*) at the beginning of the metadata field before the first space. The id is unique to the whole file and often identifies the associated sequence uniquely in a database (see section Sequence Identifiers on the Wikipedia FASTA format page).

While not documented anywhere explicitly, **only the characters up to the first space are used as identifiers** by widely used tools such as BWA. Only the identifier is carried over into files generated from the input files (BWA uses the sequence id from the genome FASTA to identify the contig/chromosome and the read id as the read name in the SAM output).

How Does It Work?

There are two requirements that a FASTA file has to fulfill to work with the FAI scheme: For each sequence in the FASTA file, **the number of characters and the number of bytes per line has to be the same**. The first restriction speaks for itself, the second restriction means that the same line ending character has to be used and no line should contain any additional spaces.

The index file then stores records of the sequence identifier, the length, the offset of the first sequence character in the file, the number of characters per line, and the number of bytes per line. With this information, we can easily compute the byte offset of the i-th character of a sequence in a file by looking at its index record. We skip to this byte offset in the file and from there, we can read the necessary sequence characters.

Building the Index

The class FaiIndex allows for building and loading FAI indices. To build such an index, we use the function build of the class FaiIndex. The first parameter is the FaiIndex object, the second is the path to the FASTA file. The function returns a `bool` indicating whether the mapping was successful (`true` on success, `false` on failure).

```
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

using namespace seqan;

int main()
{
    CharString pathToFile = getAbsolutePath("/demos/tutorial/indexed_fasta_io/example.
↪fasta");
    FaiIndex faiIndex;

    if (!build(faiIndex, toCString(pathToFile)))
        std::cout << "ERROR: Could not build the index!\n";
```

There is an alternative variant of this function where you can pass the path to the FAI file that is to be built as third parameter. The FAI file name will be stored in the FaiIndex.

```
    CharString pathToFaiFile = pathToFile;
    append(pathToFaiFile, ".fai");
    if (!build(faiIndex, toCString(pathToFile), toCString(pathToFaiFile)))
        std::cout << "ERROR: Could not build the index!\n";
```

We can write out the index after building it using the function save:

```
    if (!save(faiIndex, toCString(pathToFaiFile)))
        std::cout << "ERROR: Could not save the index to file!\n";
```

Assignment 1

Building a FAI index

Type Application

Objective Write a small program `build_fai` that takes one parameter from the command line, the path to a FASTA file. The program should then build a FAI index and write it out.

Hints Using the two-parameter variant of build is good enough.

Solution

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>

using namespace seqan;
```

```
int main(int argc, char const ** argv)
{
    if (argc != 2)
    {
        std::cerr << "USAGE: build_fai FILE.fa\n";
        return 0;
    }

    FaiIndex faiIndex;
    if (!build(faiIndex, argv[1]))
    {
        std::cerr << "ERROR: Could not build FAI index for file " << argv[1] << ".
↪\n";
        return 0;
    }

    CharString faiFilename = argv[1];
    append(faiFilename, ".fai");

    if (!save(faiIndex, toCString(faiFilename)))
    {
        std::cerr << "ERROR: Could not write the index to file!\n";
        return 0;
    }

    std::cout << "Index file " << faiFilename << " was successfully created.\n";
    return 0;
}
```

Using the Index

To load a FAI file, we use the function open: We pass the FaiIndex object as the first and the path to the FASTA file as the second parameter. The function returns a `bool` indicating whether the mapping was successful (`true` on success, `false` on failure).

```
#include <seqan/seq_io.h>
#include <seqan/sequence.h>

using namespace seqan;

int main()
{
    CharString pathToFile = getAbsolutePath("/demos/tutorial/indexed_fasta_io/example.
↪fasta");
    FaiIndex faiIndex;

    if (!open(faiIndex, toCString(pathToFile)))
        std::cout << "ERROR: Could not load FAI index " << pathToFile << ".fai\n";
```

In the example above, the FAI file `"/demos/tutorial/indexed_fasta_io/example.fasta.fai"` would be loaded. Optionally, we can specify an extra path to the FAI file:

```
    if (!open(faiIndex, toCString(pathToFile), toCString(pathToFaiFile)))
        std::cout << "ERROR: Could not load FAI index " << pathToFaiFile << "\n";
```

After loading the index, we can then use the index to map a sequence id to its (zero-based) position (a position *i* meaning that it is the *i*-th sequence) in the FASTA file using getIdByName. The function gets the FaiIndex to use, the id of the sequence, and an `unsigned` position as parameters. It returns a `bool` indicating whether the mapping was successful (`true` on success, `false` on failure).

```
    unsigned idx = 0;
    if (!getIdByName(idx, faiIndex, "chr1"))
        std::cout << "ERROR: FAI index has no entry for chr1.\n";
```

Once we have the index for the sequence in the FASTA file, we can then query the FaiIndex for the length of the sequence using sequenceLength, get the whole sequence using readSequence, or get just a part of the sequence using readRegion.

```
    unsigned seqLength = sequenceLength(faiIndex, idx);

    // Load first 10 characters of chr1.
    CharString seqChr1Prefix;
    readRegion(seqChr1Prefix, faiIndex, idx, 0, 10);

    // Load all of chr1.
    CharString seqChr1;
    readSequence(seqChr1, faiIndex, idx);
    return 0;
}
```

The sequence length can be determined by only looking at the index. When loading the sequence or a sequence infix, only the relevant part of the file will be touched. Thus, only the minimal amount of memory, time, and disk I/O is used.

Assignment 2

Using the FAI index

Type Application

Objective Write a small program `query_fai` that takes four parameters from the command line: A path to a FASTA file, the id of the sequence, a begin and an end position. The program should then read the given infix of the given sequence from the file and print it to stdout.

Hint Use the function lexicalCast to convert strings of numbers into integers.

Solution The program appears to be very long, but most is error handling, as usual with robust I/O code.

```
#include <iostream>
#include <seqan/sequence.h>
#include <seqan/seq_io.h>
#include <seqan/stream.h>

using namespace seqan;

int main(int argc, char const ** argv)
{
    if (argc != 5)
    {
        std::cerr << "USAGE: query_fai FILE.fa SEQ BEGIN END\n";
        return 0;
    }
```

```
    // Try to load index and create on the fly if necessary.
    FaiIndex faiIndex;
    if (!open(faiIndex, argv[1]))
    {
        if (!build(faiIndex, argv[1]))
        {
            std::cerr << "ERROR: Index could not be loaded or built.\n";
            return 0;
        }
        if (!save(faiIndex))    // Name is stored from when reading.
        {
            std::cerr << "ERROR: Index could not be written do disk.\n";
            return 0;
        }
    }

    // Translate sequence name to index.
    unsigned idx = 0;
    if (!getIdByName(idx, faiIndex, argv[2]))
    {
        std::cerr << "ERROR: Index does not know about sequence " << argv[2] <<
↪"\n";
        return 0;
    }

    // Convert positions into integers.
    unsigned beginPos = 0, endPos = 0;
    if (!lexicalCast(beginPos, argv[3]))
    {
        std::cerr << "ERROR: Cannot cast " << argv[3] << " into an unsigned.\n";
        return 0;
    }
    if (!lexicalCast(endPos, argv[4]))
    {
        std::cerr << "ERROR: Cannot cast " << argv[4] << " into an unsigned.\n";
        return 0;
    }

    // Make sure begin and end pos are on the sequence and begin <= end.
    if (beginPos > sequenceLength(faiIndex, idx))
        beginPos = sequenceLength(faiIndex, idx);
    if (endPos > sequenceLength(faiIndex, idx))
        endPos = sequenceLength(faiIndex, idx);
    if (beginPos > endPos)
        endPos = beginPos;

    // Finally, get infix of sequence.
    Dna5String sequenceInfix;
    readRegion(sequenceInfix, faiIndex, idx, beginPos, endPos);
    std::cout << sequenceInfix << "\n";

    return 0;
}
```

Next Steps

- Read the Wikipedia articles about the FASTA file format and the FASTQ file format and quality values to refresh your knowledge.
- Read the API documentation of the GenomicRegion class for storing regions (sequence identifier, start and end position). There also is functionality for parsing strings like `chr1:2,032-3,212` into GenomicRegion objects.
- Continue with the *Tutorials*.

ToC

Contents

Blast I/O

Learning Objective In this tutorial, you will learn about different the Blast file formats and how to interact with them in SeqAn.

Difficulty Average

Duration 1h30min - 2h30min

Prerequisite Tutorials *Sequences*, *File I/O Overview*, *Alignment*, *Pairwise Sequence Alignment*

Other recommended reading Basics of Blast Statistics

Technical requirements Full C++11 support required in compiler (GCC $\geq$ 4.9, Clang $\geq$ 3.4 or MSVC $\geq$ 2015)

The Basic local alignment search tool (BLAST) by the NCBI is one of the most widely used tools in Bioinformatics. It supports a variety of formats which, although widely used, are not standardized and partly even declared as "subject to unannounced and undocumented change". This makes implementing them very hard and is one of the reasons dealing with Blast IO is more difficult than with other file formats.

Furthermore it is important to distinguish between the Blast version written in C (known by its *blastall* executable) and the C++ version (individual *blastn*, *blastp*... executables), also known as BLAST+. The formats and their identifiers changed between versions. The following table gives an overview:

Description	*blastall*	*blast**
pairwise	-m 0	-outfmt 0
query-anchored showing identities	-m 1	-outfmt 1
query-anchored no identities	-m 2	-outfmt 2
flat query-anchored, show identities	-m 3	-outfmt 3
flat query-anchored, no identities	-m 4	-outfmt 4
query-anchored no identities and blunt ends	-m 5	
flat query-anchored, no identities and blunt ends	-m 6	
XML Blast output	-m 7	-outfmt 5
tabular	-m 8	-outfmt 6
tabular with comment lines	-m 9	-outfmt 7
Text ASN.1	-m 10	-outfmt 8
Binary ASN.1	-m 11	-outfmt 9
Comma-separated values		-outfmt 10
BLAST archive format (ASN.1)		-outfmt 11

The files written by *blastall* are considered the "legacy"-format in SeqAn. SeqAn has support for the following formats:

Format	read support	write support	based on version
pairwise		✓	Blast-2.2.26+
tabular	✓	✓	Blast-2.2.26+
tabular (legacy)	✓	✓	Blast-2.2.26
tabular w comments	✓	✓	Blast-2.2.26+
tabular w comments (legacy)	✓	✓	Blast-2.2.26

Caution: Please note that *Blast-2.2.26+* is **not the same** as *Blast-2.2.26*! One is version 2.2.26 of the C++ application suite (BLAST+) and the other is version 2.2.26 of the legacy application suite. There still are software releases for both generations.

There are different *program modes* in Blast which also influence the file format. In the legacy application suite these where specified with the `-p` parameter, in the BLAST+ suite they each have their own executable.

Program mode	query alphabet	subject alphabet
BlastN	nucleotide	nucleotide
BlastP	protein	protein
BlastX	translated nucl.	protein
TBlastN	protein	translated nucl.
TBlastX	translated nucl.	translated nucl.

Tabular formats

The tabular formats are tab-seperated-value formats (TSV), with twelve columns by default. Each line represents one match (or *high scoring pair* in Blast terminology). The twelve default columns are:

1. Query sequence ID (truncated at first whitespace)
2. Subject sequence ID (truncated at first whitespace)
3. Percentage of identical positions
4. Alignment length
5. Number of mismatches
6. Number of gap openings

7. Start position of alignment on query sequence
8. End position of alignment on query sequence
9. Start position of alignment on subject sequence
10. End position of alignment on subject sequence
11. Expect value (length normalized bit score)
12. Bit score (statistical significance indicator)

Note: Alignment positions in Blast

1. **Interval notation:** Blast uses 1-based closed intervals for positions, i.e. a match from the 100th position to the 200th position of a sequence will be shown as `100 200` in the file. SeqAn internally uses 0-based half open intervals, i.e. it starts counting at position 0 and stores the first position behind the sequence as "end", e.g. position `99` and `200` for our example.
2. **Reverse strands:** For matches found on the reverse complement strand the positions are counted backwards from the end of the sequence, e.g. a match from the 100th position to the 200th position on a reverse complement strand of a sequence of length 500 will be shown as `400 300` in the file.
3. **Translation frames:** Positions given in the file are always on the original untranslated sequence!

The `writeRecord()` function automatically does all of these conversions!

A **tabular** file could look like this (matches per query are sorted by e-value):

```
SHAA004TF        sp|P0A916|OMPW_SHIFL        50.
↪43        115        49        2        389        733        1        107        1e-
↪26        108
SHAA004TF        sp|P0A915|OMPW_ECOLI        50.
↪43        115        49        2        389        733        1        107        1e-
↪26        108
SHAA004TF        sp|P17266|OMPW_VIBCH        52.
↪21        113        45        2        410        733        4        112        5e-
↪26        106
SHAA004TF        sp|Q8ZP50|OMPW_SALTY        50.
↪43        115        49        2        389        733        1        107        3e-
↪24        101
SHAA004TF        sp|Q8Z7E2|OMPW_SALTI        50.
↪43        115        49        2        389        733        1        107        3e-
↪24        101
SHAA004TF        sp|P19766|INSB_SHISO        100.
↪00        18        0        0        803        750        114        131        4e-
↪04        43.1
SHAA004TF        sp|P19765|INSB_SHIFL        100.
↪00        18        0        0        803        750        114        131        4e-
↪04        43.1
SHAA004TF        sp|P03831|INBD_SHIDY        100.
↪00        18        0        0        803        750        114        131        4e-
↪04        43.1
SHAA004TF        sp|P59843|INSB_HAEDU        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF31|INSB_ECOLX        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF30|INSB8_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
```

```
SHAA004TF        sp|P0CF29|INSB6_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF28|INSB5_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P57998|INSB4_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF25|INSB1_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF27|INSB3_ECOLI        94.
↪44        18        1        0        803        750        150        167        0.
↪004        40.8
SHAA004TF        sp|P0CF26|INSB2_ECOLI        94.
↪44        18        1        0        803        750        150        167        0.
↪004        40.8
SHAA004TR        sp|Q0HTA5|META_SHESR        100.
↪00        77        0        0        232        2        1        77        3e-
↪42        152
SHAA004TR        sp|Q0HGZ8|META_SHESM        100.
↪00        77        0        0        232        2        1        77        3e-
↪42        152
```

The **tabular with comment lines** format additionally prefixes every block belonging to one query sequence with comment lines that include the program version, the database name and column labels. The above example would look like this:

```
# BLASTX 2.2.26+
# Query: SHAA003TR  Sample 1 Mate SHAA003TF trimmed_to 17 935
# Database: /tmp/uniprot_sprot.fasta
# 0 hits found
# BLASTX 2.2.26+
# Query: SHAA004TF  Sample 1 Mate SHAA004TR trimmed_to 25 828
# Database: /tmp/uniprot_sprot.fasta
# Fields: query id, subject id, % identity, alignment length, mismatches, gap opens,␣
↪q. start, q. end, s. start, s. end, evalue, bit score
# 17 hits found
SHAA004TF        sp|P0A916|OMPW_SHIFL        50.
↪43        115        49        2        389        733        1        107        1e-
↪26        108
SHAA004TF        sp|P0A915|OMPW_ECOLI        50.
↪43        115        49        2        389        733        1        107        1e-
↪26        108
SHAA004TF        sp|P17266|OMPW_VIBCH        52.
↪21        113        45        2        410        733        4        112        5e-
↪26        106
SHAA004TF        sp|Q8ZP50|OMPW_SALTY        50.
↪43        115        49        2        389        733        1        107        3e-
↪24        101
SHAA004TF        sp|Q8Z7E2|OMPW_SALTI        50.
↪43        115        49        2        389        733        1        107        3e-
↪24        101
SHAA004TF        sp|P19766|INSB_SHISO        100.
↪00        18        0        0        803        750        114        131        4e-
↪04        43.1
SHAA004TF        sp|P19765|INSB_SHIFL        100.
↪00        18        0        0        803        750        114        131        4e-
↪04        43.1
```

```
SHAA004TF        sp|P03831|INBD_SHIDY        100.
↪00        18        0        0        803        750        114        131        4e-
↪04        43.1
SHAA004TF        sp|P59843|INSB_HAEDU        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF31|INSB_ECOLX        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF30|INSB8_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF29|INSB6_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF28|INSB5_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P57998|INSB4_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF25|INSB1_ECOLI        100.
↪00        18        0        0        803        750        150        167        6e-
↪04        43.1
SHAA004TF        sp|P0CF27|INSB3_ECOLI        94.
↪44        18        1        0        803        750        150        167        0.
↪004        40.8
SHAA004TF        sp|P0CF26|INSB2_ECOLI        94.
↪44        18        1        0        803        750        150        167        0.
↪004        40.8
# BLASTX 2.2.26+
# Query: SHAA004TR  Sample 1 Mate SHAA004TF trimmed_to 20 853
# Database: /tmp/uniprot_sprot.fasta
# Fields: query id, subject id, % identity, alignment length, mismatches, gap opens,␣
↪q. start, q. end, s. start, s. end, evalue, bit score
# 2 hits found
SHAA004TR        sp|Q0HTA5|META_SHESR        100.
↪00        77        0        0        232        2        1        77        3e-
↪42        152
SHAA004TR        sp|Q0HGZ8|META_SHESM        100.
↪00        77        0        0        232        2        1        77        3e-
↪42        152
```

As you can see, comment lines are also printed for query sequences which don't have any matches. The major difference of these formats in BLAST+ vs the legacy application are that the *mismatches* column used to include the number of gap characters, but it does not in BLAST+. The comments also look slightly different in the **tabular with comment lines (legacy)** format:

```
# BLASTX 2.2.26 [Sep-21-2011]
# Query: SHAA003TR  Sample 1 Mate SHAA003TF trimmed_to 17 935
# Database: /tmp/uniprot_sprot.fasta
# Fields: Query id, Subject id, % identity, alignment length, mismatches, gap␣
↪openings, q. start, q. end, s. start, s. end, e-value, bit score
# BLASTX 2.2.26 [Sep-21-2011]
# Query: SHAA004TF  Sample 1 Mate SHAA004TR trimmed_to 25 828
# Database: /tmp/uniprot_sprot.fasta
# Fields: Query id, Subject id, % identity, alignment length, mismatches, gap␣
↪openings, q. start, q. end, s. start, s. end, e-value, bit score
```

```
SHAA004TF	sp|P0A916|OMPW_SHIFL	50.
↪43	115	57	2	389	733	1	107	1e-
↪26	108
SHAA004TF	sp|P0A915|OMPW_ECOLI	50.
↪43	115	57	2	389	733	1	107	1e-
↪26	108
SHAA004TF	sp|P17266|OMPW_VIBCH	52.
↪21	113	49	2	410	733	4	112	5e-
↪26	106
SHAA004TF	sp|Q8ZP50|OMPW_SALTY	50.
↪43	115	57	2	389	733	1	107	3e-
↪24	101
SHAA004TF	sp|Q8Z7E2|OMPW_SALTI	50.
↪43	115	57	2	389	733	1	107	3e-
↪24	101
SHAA004TF	sp|P19766|INSB_SHISO	100.
↪00	18	0	0	803	750	114	131	4e-
↪04	43.1
SHAA004TF	sp|P19765|INSB_SHIFL	100.
↪00	18	0	0	803	750	114	131	4e-
↪04	43.1
SHAA004TF	sp|P03831|INBD_SHIDY	100.
↪00	18	0	0	803	750	114	131	4e-
↪04	43.1
SHAA004TF	sp|P59843|INSB_HAEDU	100.
↪00	18	0	0	803	750	150	167	6e-
↪04	43.1
SHAA004TF	sp|P0CF31|INSB_ECOLX	100.
↪00	18	0	0	803	750	150	167	6e-
↪04	43.1
SHAA004TF	sp|P0CF30|INSB8_ECOLI	100.
↪00	18	0	0	803	750	150	167	6e-
↪04	43.1
SHAA004TF	sp|P0CF29|INSB6_ECOLI	100.
↪00	18	0	0	803	750	150	167	6e-
↪04	43.1
SHAA004TF	sp|P0CF28|INSB5_ECOLI	100.
↪00	18	0	0	803	750	150	167	6e-
↪04	43.1
SHAA004TF	sp|P57998|INSB4_ECOLI	100.
↪00	18	0	0	803	750	150	167	6e-
↪04	43.1
SHAA004TF	sp|P0CF25|INSB1_ECOLI	100.
↪00	18	0	0	803	750	150	167	6e-
↪04	43.1
SHAA004TF	sp|P0CF27|INSB3_ECOLI	94.
↪44	18	1	0	803	750	150	167	0.
↪004	40.8
SHAA004TF	sp|P0CF26|INSB2_ECOLI	94.
↪44	18	1	0	803	750	150	167	0.
↪004	40.8
# BLASTX 2.2.26 [Sep-21-2011]
# Query: SHAA004TR  Sample 1 Mate SHAA004TF trimmed_to 20 853
# Database: /tmp/uniprot_sprot.fasta
# Fields: Query id, Subject id, % identity, alignment length, mismatches, gap␣
↪openings, q. start, q. end, s. start, s. end, e-value, bit score
SHAA004TR	sp|Q0HTA5|META_SHESR	100.
↪00	77	0	0	232	2	1	77	3e-
↪42	152
```

```
SHAA004TR        sp|Q0HGZ8|META_SHESM        100.
↪00        77        0        0        232        2        1        77        3e-
↪42        152
```

Pairwise format

The pairwise format is the default format in Blast. It is more verbose than the tabular formats and very human readable.

This is what the last record from above would look like (the other queries are omitted):

```
BLASTX 2.2.26+


Reference: Stephen F. Altschul, Thomas L. Madden, Alejandro A.
Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J.
Lipman (1997), "Gapped BLAST and PSI-BLAST: a new generation of
protein database search programs", Nucleic Acids Res. 25:3389-3402.



Database: /foo/bar/uniprot_sprot.fasta
           538,259 sequences; 191,113,170 total letters

[...]

Query= SHAA004TR  Sample 1 Mate SHAA004TF trimmed_to 20 853

Length=833
                                                                      Score     E
Sequences producing significant alignments:                          (Bits)  Value

  sp|Q0HTA5|META_SHESR Homoserine O-succinyltransferase OS=Shewan...   152    3e-42
  sp|Q0HGZ8|META_SHESM Homoserine O-succinyltransferase OS=Shewan...   152    3e-42


> sp|Q0HTA5|META_SHESR Homoserine O-succinyltransferase OS=Shewanella
sp. (strain MR-7) GN=metA PE=3 SV=1
Length=313

 Score =  152 bits (385),  Expect = 3e-42
 Identities = 77/77 (100%), Positives = 77/77 (100%), Gaps = 0/77 (0%)
 Frame = -2

Query  232  MPVKIPDHLPAAGILESENIFVMSETRAANQDIRPMKVLILNLMPNKIETETQLLRLLGN  53
            MPVKIPDHLPAAGILESENIFVMSETRAANQDIRPMKVLILNLMPNKIETETQLLRLLGN
Sbjct  1    MPVKIPDHLPAAGILESENIFVMSETRAANQDIRPMKVLILNLMPNKIETETQLLRLLGN  60

Query  52   TPLQVDVDLLRIHDKES  2
            TPLQVDVDLLRIHDKES
Sbjct  61   TPLQVDVDLLRIHDKES  77


> sp|Q0HGZ8|META_SHESM Homoserine O-succinyltransferase OS=Shewanella
sp. (strain MR-4) GN=metA PE=3 SV=1
Length=313

 Score =  152 bits (385),  Expect = 3e-42
```

```
 Identities = 77/77 (100%), Positives = 77/77 (100%), Gaps = 0/77 (0%)
 Frame = -2

Query  232  MPVKIPDHLPAAGILESENIFVMSETRAANQDIRPMKVLILNLMPNKIETETQLLRLLGN  53
            MPVKIPDHLPAAGILESENIFVMSETRAANQDIRPMKVLILNLMPNKIETETQLLRLLGN
Sbjct  1    MPVKIPDHLPAAGILESENIFVMSETRAANQDIRPMKVLILNLMPNKIETETQLLRLLGN  60

Query  52   TPLQVDVDLLRIHDKES  2
            TPLQVDVDLLRIHDKES
Sbjct  61   TPLQVDVDLLRIHDKES  77



Lambda     K      H
   0.318    0.134    0.401

Gapped
Lambda     K      H
   0.267   0.0410    0.140

Effective search space used: 20716695286


[...]
```

Blast formats in SeqAn

There are three blast format related tags in SeqAn:

1. BlastReport for the pairwise format with the FormattedFile output specialization BlastReportFileOut.
2. BlastTabular for the tabular formats with the FormattedFile output and input specializations BlastTabularFileOut and BlastTabularFileIn.
3. BlastTabularLL which provides light-weight, but very basic tabular IO.

The third tag can be used for simple file manipulation, e.g. filtering or column rearrangement, it is not covered in this tutorial (see the dox for a simple example).

To work with the first two formats you need to understand at least the following data structures:

- BlastRecord: the record covers all BlastMatch es belonging to one query sequence.
- FormattedFile: one of BlastReportFileOut, BlastTabularFileOut and BlastTabularFileIn.
- BlastIOContext: the context of the FormattedFile.

The context contains file-global data like the name of the database and can also be used to read/write certain file format properties, e.g. “with comment lines” or “legacyFormat”.

> **Caution:** Due to the structure of blast tabular files lots of information is repeated in every block of comment lines, e.g. the database name. Because it is expected that these stay the same they are saved in the context and not the record. You may still, however, check every time you `readRecord()` if you want to make sure.

File reading example

Only tabular formats are covered in this example, because no input support is available for the pairwise format.

Copy the contents of the **tabular with comment lines** example above into a file and give it to the following program as the only parameter. Please use `.m9` as file type extension.

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char **argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_IN\n";
      return 0;
    }

    typedef Gaps<String<AminoAcid>, ArrayGaps> TGaps;
    typedef BlastMatch<TGaps, TGaps> TBlastMatch;
    typedef BlastRecord<TBlastMatch> TBlastRecord;
              << "Database: "
              << context(file).dbName
              << "\n\n";

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_IN\n";
    return 0;
```

Assignment 1

Objective Complete the above example by reading the file according to BlastTabularFileIn. For every record print the query ID, the number of contained matches and the bit-score of the best match.

Solution

Top

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char **argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_IN\n";
      return 0;
    }
```

```
    typedef Gaps<String<AminoAcid>, ArrayGaps> TGaps;
    typedef BlastMatch<TGaps, TGaps> TBlastMatch;
    typedef BlastRecord<TBlastMatch> TBlastRecord;
```

New code

```
    typedef BlastIOContext<> TContext;

    BlastTabularFileIn<TContext> file(argv[1]);

    readHeader(file);

    TBlastRecord record;

    while (onRecord(file))
    {
        // read the record
        readRecord(record, file);

        // print some diagnostics
        std::cout << "Record of query sequence \"" << record.qId << "\"\n"
                  << "=========================================\n"
            std::cout << "\n\n";
        }

        std::cout << "\n\n";
```

Bottom

```
                << "Database: "
                << context(file).dbName
                << "\n\n";

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_IN\n";
    return 0;
```

Assignment 2

Objective Study the documentation of BlastIOContext. How can you adapt the previous program to check if there were any problems reading a record? If you have come up with a solution, try to read the file at `tests/blast/defaultfields.m9`. What does the program print and why?

Solution

Top

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;
```

```
int main(int argc, char **argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_IN\n";
      return 0;
    }

    typedef Gaps<String<AminoAcid>, ArrayGaps> TGaps;
    typedef BlastMatch<TGaps, TGaps> TBlastMatch;
    typedef BlastRecord<TBlastMatch> TBlastRecord;
    typedef BlastIOContext<> TContext;

    BlastTabularFileIn<TContext> file(argv[1]);

    readHeader(file);

    TBlastRecord record;

    while (onRecord(file))
    {
        // read the record
        readRecord(record, file);

        // print some diagnostics
        std::cout << "Record of query sequence \"" << record.qId << "\"\n"
                  << "=========================================\n"
```

New code

```
        for (auto field : context(file).fields)
            std::cout << BlastMatchField<>::optionLabels[(int)field] << " ";
        std::cout << "\n\n";

        // if there is anything unexpected, tell the user about it
        if (!empty(context(file).conformancyErrors))
        {
            std::cout << "There were non critical errors when reading the␣
↪record:\n";
            write(std::cout, context(file).conformancyErrors);
            std::cout << "\n\n";
        }

        if (!empty(context(file).otherLines))
        {
            std::cout << "There were unidentified lines in the comments:\n";
            write(std::cout, context(file).otherLines);
            std::cout << "\n\n";
        }

        std::cout << "\n\n";
```

The program will print conformancyErrors for the last record, because there is a typo in the file (`Datacase` instead of `Database`).

Bottom

```
                << "Database: "
                << context(file).dbName
                << "\n\n";

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_IN\n";
    return 0;
```

Assignment 3

Objective Now that you have a basic understanding of BlastIOContext, also print the following information after reading the records:

- file format (with comment lines or without, BLAST+ or legacy?)
- blast program and version
- name of database

Verify that the results are as expected on the files `tests/blast/defaultfields.m8`, `tests/blast/defaultfields.m9` and `tests/blast/defaultfields_legacy.m9`.

Solution

Top

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char **argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_IN\n";
      return 0;
    }

    typedef Gaps<String<AminoAcid>, ArrayGaps> TGaps;
    typedef BlastMatch<TGaps, TGaps> TBlastMatch;
    typedef BlastRecord<TBlastMatch> TBlastRecord;
    typedef BlastIOContext<> TContext;

    BlastTabularFileIn<TContext> file(argv[1]);

    readHeader(file);

    TBlastRecord record;

    while (onRecord(file))
    {
```

```
        // read the record
        readRecord(record, file);

        // print some diagnostics
        std::cout << "Record of query sequence \"" << record.qId << "\"\n"
                  << "========================================\n"
        for (auto field : context(file).fields)
            std::cout << BlastMatchField<>::optionLabels[(int)field] << " ";
        std::cout << "\n\n";

        // if there is anything unexpected, tell the user about it
        if (!empty(context(file).conformancyErrors))
        {
            std::cout << "There were non critical errors when reading the␣
↪record:\n";
            write(std::cout, context(file).conformancyErrors);
            std::cout << "\n\n";
        }

        if (!empty(context(file).otherLines))
        {
            std::cout << "There were unidentified lines in the comments:\n";
            write(std::cout, context(file).otherLines);
            std::cout << "\n\n";
        }

        std::cout << "\n\n";
```

New code

```
    }

    readFooter(file);

    std::cout << "File Format: tabular"
              << (context(file).tabularSpec == BlastTabularSpec::COMMENTS ? "␣
↪with coment lines" : "")
              << '\n'
              << "Generation: "
              << (context(file).legacyFormat ? " legacy" : " BLAST+")
              << '\n'
              << "Program and version: "
              << context(file).versionString
              << '\n'
```

Bottom

```
              << "Database: "
              << context(file).dbName
              << "\n\n";

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_IN\n";
    return 0;
```

Assignment 4

As previously mentioned, twelve columns are printed by default. This can be changed in BLAST+, also by means of the `--outfmt` parameter. A standards compliant **file with comment lines** and custom column composition can be read without further configuration in SeqAn.

Tip: Don't believe it? Look at `tests/blast/customfields.m9`, as as you can see the bit score is in the 13th column (instead of the twelfth). If you run your program on this file, it should still print the correct bit-scores!

Objective Read BlastIOContext again focusing on fields and also read BlastMatchField. Now adapt the previous program to print for every record the `optionLabel` of each field used.

Verify that the results are as expected on the files `tests/blast/defaultfields.m9` and `tests/blast/customfields.m9`.

Solution

Top

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char **argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_IN\n";
      return 0;
    }

    typedef Gaps<String<AminoAcid>, ArrayGaps> TGaps;
    typedef BlastMatch<TGaps, TGaps> TBlastMatch;
    typedef BlastRecord<TBlastMatch> TBlastRecord;
    typedef BlastIOContext<> TContext;

    BlastTabularFileIn<TContext> file(argv[1]);

    readHeader(file);

    TBlastRecord record;

    while (onRecord(file))
    {
        // read the record
        readRecord(record, file);

        // print some diagnostics
        std::cout << "Record of query sequence \"" << record.qId << "\"\n"
                  << "=========================================\n"
```

New code

```
                  << "Number of HSPs: " << length(record.matches) << "\n";
        if  (!empty(record.matches))
            std::cout << "BitScore of best HSP: " << front(record.matches).
↪bitScore << "\n";

        // print column composition
        std::cout << "Columns: ";
```

Bottom

```
        for (auto field : context(file).fields)
            std::cout << BlastMatchField<>::optionLabels[(int)field] << " ";
        std::cout << "\n\n";

        // if there is anything unexpected, tell the user about it
        if (!empty(context(file).conformancyErrors))
        {
            std::cout << "There were non critical errors when reading the␣
↪record:\n";
            write(std::cout, context(file).conformancyErrors);
            std::cout << "\n\n";
        }

        if (!empty(context(file).otherLines))
        {
            std::cout << "There were unidentified lines in the comments:\n";
            write(std::cout, context(file).otherLines);
            std::cout << "\n\n";
        }

        std::cout << "\n\n";
    }

    readFooter(file);

    std::cout << "File Format: tabular"
              << (context(file).tabularSpec == BlastTabularSpec::COMMENTS ? "␣
↪with coment lines" : "")
              << '\n'
              << "Generation: "
              << (context(file).legacyFormat ? " legacy" : " BLAST+")
              << '\n'
              << "Program and version: "
              << context(file).versionString
              << '\n'
              << "Database: "
              << context(file).dbName
              << "\n\n";

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_IN\n";
    return 0;
```

If this was too easy, you can also try the same for tabular files without comment lines!

File writing example

The following program stub creates three query sequences and two subject sequences in amino acid alphabet. We will later generate records with matches and print these to disk.

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char ** argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_OUT\n";
      return 0;
    }

    typedef String<AminoAcid>               TSequence;
    typedef std::string                     TId;
    typedef Gaps<TSequence, ArrayGaps>      TGaps;
    typedef BlastMatch<TGaps, TGaps>        TBlastMatch;
    typedef BlastRecord<TBlastMatch>        TBlastRecord;
    typedef BlastIOContext<Blosum62>        TContext;

    StringSet<TSequence>    queries;
    StringSet<TSequence>    subjects;
    StringSet<TId>          qIds;
    StringSet<TId>          sIds;

    appendValue(queries, "VAYAQPRKLCYP");
    appendValue(queries, "NAYPRUTEIN");
    appendValue(queries, "AVITSFTQ");

    appendValue(subjects,
        "SSITEEKHIPHKEQDKDAEFLSKEALKTHMTENVLQMDRRAVQDPSTSFLQLLKAKGLLG"
        "LPDYEVNLADVNSPGFRKVAYAQTKPRRLCFPNGGTRRGSFIMDTAVVVMVSLRYVNIGK"
        "VIFPGATDVSEGEDEFWAGLPQAYGCLATEFLCIHIAIYSWIHVQSSRYDDMNASVIRAK"
        "LNLAVITSWTQLIQAEKETI");

    appendValue(subjects,
        "GATRDSKGNAVITSFTQARLRVYADLLGPYWIILHVIELTGVGNTGQKCTLNHMGTYAVF"
        "DLKQPPATNDLGLPKPCFIGFDIQNELAIGTVGHSEAVIAAFTQRDRLEERAESKQSLAR"
        "PVISPKLIAEVSTVLESALNQMYSSLGFYRVERAEDYAQPRKLCVVKKKSFNCLNADIWL"
        "EYRMEDQKSVPKVFKIMMDD");

    appendValue(qIds, "Query_Numero_Uno with args");
    appendValue(qIds, "Query_Numero_Dos with args");
    appendValue(qIds, "Query_Numero_Tres with args");

    appendValue(sIds, "Subject_Numero_Uno");
    appendValue(sIds, "Subject_Numero_Dos");
    }

    writeFooter(outfile);
```

```
    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_OUT\n";
```

Assignment 5

Objective Before we can begin to align, certain properties of the BlastIOContext have to be set. Which ones? Add a block with the necessary initialization to the above code, use blast defaults where possible. Although not strictly required at this point: include a call to `writeHeader()`.

> **Caution:** Alignment score computation works slightly different in Blast and in SeqAn, please have a look at BlastScoringScheme. For the task at hand it should suffice to simply use the corresponding `set*()` functions on scoringScheme.

Solution

Top

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char ** argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_OUT\n";
      return 0;
    }

    typedef String<AminoAcid>               TSequence;
    typedef std::string                     TId;
    typedef Gaps<TSequence, ArrayGaps>      TGaps;
    typedef BlastMatch<TGaps, TGaps>        TBlastMatch;
    typedef BlastRecord<TBlastMatch>        TBlastRecord;
    typedef BlastIOContext<Blosum62>        TContext;

    StringSet<TSequence>    queries;
    StringSet<TSequence>    subjects;
    StringSet<TId>          qIds;
    StringSet<TId>          sIds;

    appendValue(queries, "VAYAQPRKLCYP");
    appendValue(queries, "NAYPRUTEIN");
    appendValue(queries, "AVITSFTQ");

    appendValue(subjects,
        "SSITEEKHIPHKEQDKDAEFLSKEALKTHMTENVLQMDRRAVQDPSTSFLQLLKAKGLLG"
```

```
        "LPDYEVNLADVNSPGFRKVAYAQTKPRRLCFPNGGTRRGSFIMDTAVVVMVSLRYVNIGK"
        "VIFPGATDVSEGEDEFWAGLPQAYGCLATEFLCIHIAIYSWIHVQSSRYDDMNASVIRAK"
        "LNLAVITSWTQLIQAEKETI");

    appendValue(subjects,
        "GATRDSKGNAVITSFTQARLRVYADLLGPYWIILHVIELTGVGNTGQKCTLNHMGTYAVF"
        "DLKQPPATNDLGLPKPCFIGFDIQNELAIGTVGHSEAVIAAFTQRDRLEERAESKQSLAR"
        "PVISPKLIAEVSTVLESALNQMYSSLGFYRVERAEDYAQPRKLCVVKKKSFNCLNADIWL"
        "EYRMEDQKSVPKVFKIMMDD");

    appendValue(qIds, "Query_Numero_Uno with args");
    appendValue(qIds, "Query_Numero_Dos with args");
    appendValue(qIds, "Query_Numero_Tres with args");

    appendValue(sIds, "Subject_Numero_Uno");
    appendValue(sIds, "Subject_Numero_Dos");
```

New code

```
    BlastTabularFileOut<TContext> outfile(argv[1]);
    String<TBlastRecord> records;

    // protein vs protein search is BLASTP
    context(outfile).blastProgram = BlastProgram::BLASTP;

    // set gap parameters in blast notation
    setScoreGapOpenBlast(context(outfile).scoringScheme, -11);
    setScoreGapExtend(context(outfile).scoringScheme, -1);
    SEQAN_ASSERT(isValid(context(outfile).scoringScheme));

    // set the database properties in the context
    context(outfile).dbName = "The Foo Database";
    context(outfile).dbTotalLength = length(concat(subjects));
```

Bottom

```
    writeFooter(outfile);

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_OUT\n";
    return 0;
}
```

Assignment 6

Objective Next create a record for every query sequence, and in each record a match for every query-subject pair. Compute the local alignment for each of those matches. Use the align member of the match object.

Solution

Top

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char ** argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_OUT\n";
      return 0;
    }

    typedef String<AminoAcid>               TSequence;
    typedef std::string                     TId;
    typedef Gaps<TSequence, ArrayGaps>      TGaps;
    typedef BlastMatch<TGaps, TGaps>        TBlastMatch;
    typedef BlastRecord<TBlastMatch>        TBlastRecord;
    typedef BlastIOContext<Blosum62>        TContext;

    StringSet<TSequence>    queries;
    StringSet<TSequence>    subjects;
    StringSet<TId>          qIds;
    StringSet<TId>          sIds;

    appendValue(queries, "VAYAQPRKLCYP");
    appendValue(queries, "NAYPRUTEIN");
    appendValue(queries, "AVITSFTQ");

    appendValue(subjects,
        "SSITEEKHIPHKEQDKDAEFLSKEALKTHMTENVLQMDRRAVQDPSTSFLQLLKAKGLLG"
        "LPDYEVNLADVNSPGFRKVAYAQTKPRRLCFPNGGTRRGSFIMDTAVVVMVSLRYVNIGK"
        "VIFPGATDVSEGEDEFWAGLPQAYGCLATEFLCIHIAIYSWIHVQSSRYDDMNASVIRAK"
        "LNLAVITSWTQLIQAEKETI");

    appendValue(subjects,
        "GATRDSKGNAVITSFTQARLRVYADLLGPYWIILHVIELTGVGNTGQKCTLNHMGTYAVF"
        "DLKQPPATNDLGLPKPCFIGFDIQNELAIGTVGHSEAVIAAFTQRDRLEERAESKQSLAR"
        "PVISPKLIAEVSTVLESALNQMYSSLGFYRVERAEDYAQPRKLCVVKKKSFNCLNADIWL"
        "EYRMEDQKSVPKVFKIMMDD");

    appendValue(qIds, "Query_Numero_Uno with args");
    appendValue(qIds, "Query_Numero_Dos with args");
    appendValue(qIds, "Query_Numero_Tres with args");

    appendValue(sIds, "Subject_Numero_Uno");
    appendValue(sIds, "Subject_Numero_Dos");

    BlastTabularFileOut<TContext> outfile(argv[1]);
    String<TBlastRecord> records;

    // protein vs protein search is BLASTP
    context(outfile).blastProgram = BlastProgram::BLASTP;

    // set gap parameters in blast notation
    setScoreGapOpenBlast(context(outfile).scoringScheme, -11);
```

```
    setScoreGapExtend(context(outfile).scoringScheme, -1);
    SEQAN_ASSERT(isValid(context(outfile).scoringScheme));

    // set the database properties in the context
    context(outfile).dbName = "The Foo Database";
    context(outfile).dbTotalLength = length(concat(subjects));
```

New Code

```
    context(outfile).dbNumberOfSeqs = length(subjects);

    writeHeader(outfile); // write file header

    for (unsigned q = 0; q < length(queries); ++q)
    {
        appendValue(records, TBlastRecord(qIds[q]));
        TBlastRecord & r = back(records);

        r.qLength = length(queries[q]);

        for (unsigned s = 0; s < length(subjects); ++s)
        {
            appendValue(r.matches, TBlastMatch(qIds[q], sIds[s]));
            TBlastMatch & m = back(records[q].matches);

            assignSource(m.alignRow0, queries[q]);
            assignSource(m.alignRow1, subjects[s]);

            if (m.eValue > 1)
                eraseBack(records[q].matches);
```

Bottom

```
    writeFooter(outfile);

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_OUT\n";
    return 0;
}
```

Assignment 7

Objective Now that you have the align member computed for every match, also save the begin and end positions, as well as the lengths. Blast Output needs to now about the number of gaps, mismatches... of every match, how can they be computed?

Solution

Top

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
```

```
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char ** argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_OUT\n";
      return 0;
    }

    typedef String<AminoAcid>               TSequence;
    typedef std::string                     TId;
    typedef Gaps<TSequence, ArrayGaps>      TGaps;
    typedef BlastMatch<TGaps, TGaps>        TBlastMatch;
    typedef BlastRecord<TBlastMatch>        TBlastRecord;
    typedef BlastIOContext<Blosum62>        TContext;

    StringSet<TSequence>    queries;
    StringSet<TSequence>    subjects;
    StringSet<TId>          qIds;
    StringSet<TId>          sIds;

    appendValue(queries, "VAYAQPRKLCYP");
    appendValue(queries, "NAYPRUTEIN");
    appendValue(queries, "AVITSFTQ");

    appendValue(subjects,
        "SSITEEKHIPHKEQDKDAEFLSKEALKTHMTENVLQMDRRAVQDPSTSFLQLLKAKGLLG"
        "LPDYEVNLADVNSPGFRKVAYAQTKPRRLCFPNGGTRRGSFIMDTAVVVMVSLRYVNIGK"
        "VIFPGATDVSEGEDEFWAGLPQAYGCLATEFLCIHIAIYSWIHVQSSRYDDMNASVIRAK"
        "LNLAVITSWTQLIQAEKETI");

    appendValue(subjects,
        "GATRDSKGNAVITSFTQARLRVYADLLGPYWIILHVIELTGVGNTGQKCTLNHMGTYAVF"
        "DLKQPPATNDLGLPKPCFIGFDIQNELAIGTVGHSEAVIAAFTQRDRLEERAESKQSLAR"
        "PVISPKLIAEVSTVLESALNQMYSSLGFYRVERAEDYAQPRKLCVVKKKSFNCLNADIWL"
        "EYRMEDQKSVPKVFKIMMDD");

    appendValue(qIds, "Query_Numero_Uno with args");
    appendValue(qIds, "Query_Numero_Dos with args");
    appendValue(qIds, "Query_Numero_Tres with args");

    appendValue(sIds, "Subject_Numero_Uno");
    appendValue(sIds, "Subject_Numero_Dos");

    BlastTabularFileOut<TContext> outfile(argv[1]);
    String<TBlastRecord> records;

    // protein vs protein search is BLASTP
    context(outfile).blastProgram = BlastProgram::BLASTP;

    // set gap parameters in blast notation
    setScoreGapOpenBlast(context(outfile).scoringScheme, -11);
    setScoreGapExtend(context(outfile).scoringScheme, -1);
    SEQAN_ASSERT(isValid(context(outfile).scoringScheme));
```

```
    // set the database properties in the context
    context(outfile).dbName = "The Foo Database";
    context(outfile).dbTotalLength = length(concat(subjects));
    context(outfile).dbNumberOfSeqs = length(subjects);

    writeHeader(outfile); // write file header

    for (unsigned q = 0; q < length(queries); ++q)
    {
        appendValue(records, TBlastRecord(qIds[q]));
        TBlastRecord & r = back(records);

        r.qLength = length(queries[q]);

        for (unsigned s = 0; s < length(subjects); ++s)
        {
            appendValue(r.matches, TBlastMatch(qIds[q], sIds[s]));
            TBlastMatch & m = back(records[q].matches);

            assignSource(m.alignRow0, queries[q]);
            assignSource(m.alignRow1, subjects[s]);
```

New Code

```
            localAlignment(m.alignRow0, m.alignRow1,␣
↪seqanScheme(context(outfile).scoringScheme));

            m.qStart = beginPosition(m.alignRow0);
            m.qEnd   = endPosition(m.alignRow0);
            m.sStart = beginPosition(m.alignRow1);
            m.sEnd   = endPosition(m.alignRow1);

            m.qLength = length(queries[q]);
            m.sLength = length(subjects[s]);
```

Bottom

```
            if (m.eValue > 1)
                eraseBack(records[q].matches);


    writeFooter(outfile);

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_OUT\n";
    return 0;
}
```

Assignment 8

Objective

Finally add e-value statistics and print the results to a file:

- compute the bit score and e-value for every match
- discard matches with an e-value greater than 1
- for every record, sort the matches by bit-score
- write each record
- write the footer before exiting the program

Solution

Top

```
#include <iostream>
#include <seqan/basic.h>
#ifndef STDLIB_VS
#include <seqan/blast.h>

using namespace seqan;

int main(int argc, char ** argv)
{
    if (argc != 2)
    {
      std::cerr << "USAGE: FILE_OUT\n";
      return 0;
    }

    typedef String<AminoAcid>               TSequence;
    typedef std::string                     TId;
    typedef Gaps<TSequence, ArrayGaps>      TGaps;
    typedef BlastMatch<TGaps, TGaps>        TBlastMatch;
    typedef BlastRecord<TBlastMatch>        TBlastRecord;
    typedef BlastIOContext<Blosum62>        TContext;

    StringSet<TSequence>    queries;
    StringSet<TSequence>    subjects;
    StringSet<TId>          qIds;
    StringSet<TId>          sIds;

    appendValue(queries, "VAYAQPRKLCYP");
    appendValue(queries, "NAYPRUTEIN");
    appendValue(queries, "AVITSFTQ");

    appendValue(subjects,
        "SSITEEKHIPHKEQDKDAEFLSKEALKTHMTENVLQMDRRAVQDPSTSFLQLLKAKGLLG"
        "LPDYEVNLADVNSPGFRKVAYAQTKPRRLCFPNGGTRRGSFIMDTAVVVMVSLRYVNIGK"
        "VIFPGATDVSEGEDEFWAGLPQAYGCLATEFLCIHIAIYSWIHVQSSRYDDMNASVIRAK"
        "LNLAVITSWTQLIQAEKETI");

    appendValue(subjects,
        "GATRDSKGNAVITSFTQARLRVYADLLGPYWIILHVIELTGVGNTGQKCTLNHMGTYAVF"
        "DLKQPPATNDLGLPKPCFIGFDIQNELAIGTVGHSEAVIAAFTQRDRLEERAESKQSLAR"
        "PVISPKLIAEVSTVLESALNQMYSSLGFYRVERAEDYAQPRKLCVVKKKSFNCLNADIWL"
        "EYRMEDQKSVPKVFKIMMDD");

    appendValue(qIds, "Query_Numero_Uno with args");
    appendValue(qIds, "Query_Numero_Dos with args");
    appendValue(qIds, "Query_Numero_Tres with args");
```

```
    appendValue(sIds, "Subject_Numero_Uno");
    appendValue(sIds, "Subject_Numero_Dos");

    BlastTabularFileOut<TContext> outfile(argv[1]);
    String<TBlastRecord> records;

    // protein vs protein search is BLASTP
    context(outfile).blastProgram = BlastProgram::BLASTP;

    // set gap parameters in blast notation
    setScoreGapOpenBlast(context(outfile).scoringScheme, -11);
    setScoreGapExtend(context(outfile).scoringScheme, -1);
    SEQAN_ASSERT(isValid(context(outfile).scoringScheme));

    // set the database properties in the context
    context(outfile).dbName = "The Foo Database";
    context(outfile).dbTotalLength = length(concat(subjects));
    context(outfile).dbNumberOfSeqs = length(subjects);

    writeHeader(outfile); // write file header

    for (unsigned q = 0; q < length(queries); ++q)
    {
        appendValue(records, TBlastRecord(qIds[q]));
        TBlastRecord & r = back(records);

        r.qLength = length(queries[q]);

        for (unsigned s = 0; s < length(subjects); ++s)
        {
            appendValue(r.matches, TBlastMatch(qIds[q], sIds[s]));
            TBlastMatch & m = back(records[q].matches);

            assignSource(m.alignRow0, queries[q]);
            assignSource(m.alignRow1, subjects[s]);

            localAlignment(m.alignRow0, m.alignRow1,␣
↪seqanScheme(context(outfile).scoringScheme));

            m.qStart = beginPosition(m.alignRow0);
            m.qEnd   = endPosition(m.alignRow0);
            m.sStart = beginPosition(m.alignRow1);
            m.sEnd   = endPosition(m.alignRow1);

            m.qLength = length(queries[q]);
            m.sLength = length(subjects[s]);
```

New Code

```
            computeAlignmentStats(m, context(outfile));
            computeBitScore(m, context(outfile));
            computeEValue(m, context(outfile));
```

Bottom

```
            if (m.eValue > 1)
                eraseBack(records[q].matches);
        }
```

```
        r.matches.sort(); // sort by bitscore

        writeRecord(outfile, r);
    }

    writeFooter(outfile);

    return 0;
}
#else
int main()
{
    std::cerr << "USAGE: FILE_OUT\n";
    return 0;
}
```

Your output file should look like this:

```
# BLASTP 2.2.26+ [I/O Module of SeqAn-2.0.0, http://www.seqan.de]
# Query: Query_Numero_Uno with args
# Database: The Foo Database
# Fields: query id, subject id, % identity, alignment length, mismatches, gap␣
↪opens, q. start, q. end, s. start, s. end, evalue, bit score
# 2 hits found
Query_Numero_Uno        Subject_Numero_Uno        71.
↪43        14        2        1        1        12        79        92        5e-
↪04        23.1
Query_Numero_Uno        Subject_Numero_Dos        100.
↪00        8        0        0        3        10        157        164        0.
↪001        22.3
# BLASTP 2.2.26+ [I/O Module of SeqAn-2.0.0, http://www.seqan.de]
# Query: Query_Numero_Dos with args
# Database: The Foo Database
# 0 hits found
# BLASTP 2.2.26+ [I/O Module of SeqAn-2.0.0, http://www.seqan.de]
# Query: Query_Numero_Tres with args
# Database: The Foo Database
# Fields: query id, subject id, % identity, alignment length, mismatches, gap␣
↪opens, q. start, q. end, s. start, s. end, evalue, bit score
# 2 hits found
Query_Numero_Tres        Subject_Numero_Dos        100.
↪00        8        0        0        1        8        10        17        0.
↪007        18.9
Query_Numero_Tres        Subject_Numero_Uno        87.
↪50        8        1        0        1        8        184        191        0.
↪026        16.9
# BLAST processed 3 queries
```

Assignment 9

Objective Up until now you have only printed the **tabular with comment lines** format. What do you have to do to print without comment lines? In legacy format? What about the pairwise format?

Solution

Tabular without comment lines Add `context(outfile).tabularSpec =`

`BlastTabularSpec::NO_COMMENTS` at l.53. Remember to use `.m8` as file extension!

The result should look like this:

```
Query_Numero_Uno        Subject_Numero_Uno        71.
↪43        14        2        1        1        12        79        92        5e-
↪04        23.1
Query_Numero_Uno        Subject_Numero_Dos        100.
↪00        8        0        0        3        10        157        164        0.
↪001        22.3
Query_Numero_Tres        Subject_Numero_Dos        100.
↪00        8        0        0        1        8        10        17        0.
↪007        18.9
Query_Numero_Tres        Subject_Numero_Uno        87.
↪50        8        1        0        1        8        184        191        0.
↪026        16.9
```

Tabular without comment lines (legacy) To print in legacy tabular format (with or without comment lines), add `context(outfile).legacyFormat = true` at l.53.

The result should look like this(legacy and NO_COMMENTS):

```
Query_Numero_Uno        Subject_Numero_Uno        71.
↪43        14        4        1        1        12        79        92        5e-
↪04        23.1
Query_Numero_Uno        Subject_Numero_Dos        100.
↪00        8        0        0        3        10        157        164        0.
↪001        22.3
Query_Numero_Tres        Subject_Numero_Dos        100.
↪00        8        0        0        1        8        10        17        0.
↪007        18.9
Query_Numero_Tres        Subject_Numero_Uno        87.
↪50        8        1        0        1        8        184        191        0.
↪026        16.9
```

Pairwise format To print in the pairwise format replace l.52 with `BlastReportFileOut<TContext> outfile(argv[1]);`.

Remember to use `.m0` as file extension!

The result should look like this:

```
BLASTP 2.2.26+ [I/O Module of SeqAn-2.0.0, http://www.seqan.de]


Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.



Reference for SeqAn: Doering, A., D. Weese, T. Rausch, K. Reinert (2008):␣
↪SeqAn --
An efficient, generic C++ library for sequence analysis. BMC Bioinformatics,
9(1), 11. BioMed Central Ltd. doi:10.1186/1471-2105-9-11



Database: The Foo Database
```

```
          2 sequences; 400 total letters


Query= Query_Numero_Uno with args

Length=12
                                                                      Score     E
Sequences producing significant alignments:                          (Bits) ␣
↪Value

Subject_Numero_Uno                                                     23   0.
↪0005
Subject_Numero_Dos                                                     22   0.
↪0009

ALIGNMENTS
> Subject_Numero_Uno
Length=200

 Score =  23.1 bits (48), Expect =  0.0005
 Identities = 10/14 (71%), Positives = 12/14 (86%), Gaps = 2/14 (14%)

Query  1   VAYAQ--PRKLCYP  12
           VAYAQ  PR+LC+P
Sbjct  79  VAYAQTKPRRLCFP  92


> Subject_Numero_Dos
Length=200

 Score =  22.3 bits (46), Expect =  0.0009
 Identities = 8/8 (100%), Positives = 8/8 (100%), Gaps = 0/8 (0%)

Query  3    YAQPRKLC  10
            YAQPRKLC
Sbjct  157  YAQPRKLC  164



Lambda     K      H
   0.267   0.0410   0.1400

Gapped
Lambda     K      H
   0.267   0.0410   0.1400

Effective search space used: 4800


Query= Query_Numero_Dos with args

Length=10


***** No hits found *****
```

```
Lambda     K      H
   0.267   0.0410   0.1400

Gapped
Lambda     K      H
   0.267   0.0410   0.1400

Effective search space used: 4000


Query= Query_Numero_Tres with args

Length=8
                                                                      Score     E
Sequences producing significant alignments:                          (Bits) ␣
↪Value

Subject_Numero_Dos                                                     18   0.007
Subject_Numero_Uno                                                     16   0.03

ALIGNMENTS
> Subject_Numero_Dos
Length=200

 Score =  18.9 bits (37), Expect =  0.007
 Identities = 8/8 (100%), Positives = 8/8 (100%), Gaps = 0/8 (0%)

Query  1   AVITSFTQ  8
           AVITSFTQ
Sbjct  10  AVITSFTQ  17


> Subject_Numero_Uno
Length=200

 Score =  16.9 bits (32), Expect =  0.03
 Identities = 7/8 (88%), Positives = 8/8 (100%), Gaps = 0/8 (0%)

Query  1    AVITSFTQ  8
            AVITS+TQ
Sbjct  184  AVITSWTQ  191



Lambda     K      H
   0.267   0.0410   0.1400

Gapped
Lambda     K      H
   0.267   0.0410   0.1400

Effective search space used: 3200


  Database: The Foo Database
  Number of letters in database: 400
  Number of sequences in database:  2
```

```
Matrix:BLOSUM62
Gap Penalties: Existence: 11, Extension: 1
```

ToC

Contents

SAM and BAM I/O

Learning Objective In this tutorial, you will learn how to read and write SAM and BAM files.

Difficulty Average

Duration 1 h (45 min if you know the SAM format)

Prerequisites *Sequences*, *File I/O Overview*, SAM Format Specification

Overview

Warning: Before you can read/write BAM files (bgzf compressed SAM files) you need to make sure that your program is linked against the zlib library. When you build your application within the SeqAn build infrastructure, the zlib library is automatically located by looking at the standard places for the library. Also have a look at *Formatted Files* to read more about support of compressed file I/O. If the macro `SEQAN_HAS_ZLIB` is set to `0` then reading/writing BAM file format is disabled. It is set to `1` if the zlib could be found and reading/writing of compressed files is enabled automatically. You can read *Using SeqAn in CMake-based projects*, *Integration with your own Build System* and *Installing Dependencies* for further notes about using the zlib and libbz2 in your build infrastructure.

This tutorial shows how to read and write SAM and BAM files using the BamFileIn and BamFileOut classes. It starts out with a quick reminder on the structure of SAM (and also BAM) files and continues with how to read and write SAM/BAM files and access the tags of a record.

Important: Note that this tutorial is targeted at readers that already know about the SAM format. If you do not know about the SAM format yet, then this tutorial will be harder for your to understand.

Both SAM and BAM files store multi-read alignments. Storing alignments of longer sequences such as contigs from assemblies is also possible, but less common. Here, we will focus on multi-read alignments.

SAM files are text files, having one record per line. BAM files are just binary, compressed versions of SAM files that have a stricter organization and aim to be more efficiently usable by programs and computers. The nuts and bolts of the formats are described in the SAM Format Specification.

The SAM and BAM related I/O functionality in SeqAn focuses on allowing access to these formats in SeqAn with thin abstractions. The *Fragment Store* Tutorial shows how to get a more high-level abstraction for multi-read alignments.

Important: SAM/BAM I/O vs. Fragment Store

The *Fragment Store* provides a high-level view of multi-read alignments. This is very useful if you want to do SNP or small indel detection because you need to access the alignment of the reads around your candidate regions. However, storing the whole alignment of a 120GB BAM file obviously is not a good idea.

The SAM/BAM I/O functionality in SeqAn is meant for sequentially reading through SAM and BAM files. Jumping within BAM files using BAI indices is described in the *Using BAM Indices* section of this tutorial.

SAM / BAM Format

The following shows an example of a SAM file.

```
@HD     VN:1.3  SO:coordinate
@SQ     SN:ref  LN:45
@SQ     SN:ref2  LN:40
r001    163     ref     7       30      8M4I4M1D3M      =       37      39      T
↪12561,2,20,112
r002    0       ref     9       30      1S2I6M1P1I1P1I4M2I      *       0       0
r003    0       ref     9       30      5H6M    *       0       0       AGCTAA
r004    0       ref     16      30      6M14N1I5M       *       0       0       ATAG
r003    16      ref     29      30      6H5M    *       0       0       TAGGC
r001    83      ref     37      30      9M      =       7       -
↪39     CAGCGCCAT       *
```

SAM files are TSV (tab-separated-values) files and begin with an optional header. The header consists of multiple lines, starting with an `'@'` character, each line is a record. Each record starts with its identifier and is followed by tab-separated tags. Each tag in the header consists of a two-character identifier, followed by `':'`, followed by the value.

If present, the `@HD` record must be the first record which specifies the SAM version (tag `VN`) used in this file and the sort order (`SO`). The optional `@SQ` header records give the reference sequence names (tag `SN`) and lengths (tag `LN`). There also are other header record types.

The optional header section is followed by the alignment records. The alignment records are again tab-separated. There are 11 mandatory columns.

Col	Field	Type	N/A Value	Description
1	QNAME	string	mandatory	The query/read name.
2	FLAG	int	mandatory	The record's flag.
3	RNAME	string	*	The reference name.
4	POS	32-bit int	0	1-based position on the reference.
5	MAPQ	8-bit int	255	The mapping quality.
6	CIGAR	string	*	The CIGAR string of the alignment.
7	RNEXT	string	*	The reference of the next mate/segment.
8	PNEXT	string	0	The position of the next mate/seqgment.
9	TLEN	string	0	The observed length of the template.
10	SEQ	string	*	The query/read sequence.
11	QUAL	string	*	The ASCII PHRED-encoded base qualities.

Notes:

- The SAM standard talks about "queries". In the context of read mapping, where the format originates, queries are reads.
- The SAM standard talks about "templates" and "segments". In the case of paired-end and mate-pair mapping the template consists of two segments, each is one read. The template length is the insert size.
- Paired-end reads are stored as two alignments records with the same QNAME. The first and second mate are discriminated by the FLAG values.
- When the FLAG indicates that SEQ is reverse-complemented, then QUAL is reversed.
- Positions in the SAM file are 1-based. When read into a BamAlignmentRecord (see below), the positions become 0-based.
- The qualities must be stored as ASCII PHRED-encoded qualities.
- The query and reference names must not contain whitespace. It is common to trim query and reference ids at the first space.

There are many ambiguities, recommendations, and some special cases in the formats that we do not describe here. We recommend that you follow this tutorial, start working with the SAM and BAM formats and later read the SAM specification "on demand" when you need it.

The 11 mandatory columns are followed by an arbitrary number of optional tags. Tags have a two-character identifier followed by `":${TYPE}:"`, followed by the tag's value.

BAM files store their header as plain-text SAM headers. However, they additionally store the name and length information about the reference sequences. This information is mandatory since in BAM, the alignment records only contain the numeric ids of the reference sequences. Thus, the name is stored outside the record in the header.

A First Working Example

The following program reads a file named `example.sam` and prints its contents back to the user on standard output.

```
#include <seqan/bam_io.h>

using namespace seqan;

int main()
{
    CharString bamFileName = getAbsolutePath("demos/tutorial/sam_and_bam_io/example.
↪sam");

    // Open input file, BamFileIn can read SAM and BAM files.
```

```
    BamFileIn bamFileIn;
    if (!open(bamFileIn, toCString(bamFileName)))
    {
        std::cerr << "ERROR: Could not open " << bamFileName << std::endl;
        return 1;
    }
    // Open output file, BamFileOut accepts also an ostream and a format tag.
    BamFileOut bamFileOut(context(bamFileIn), std::cout, Sam());

    try
    {
        // Copy header.
        BamHeader header;
        readHeader(header, bamFileIn);
        writeHeader(bamFileOut, header);

        // Copy records.
        BamAlignmentRecord record;
        while (!atEnd(bamFileIn))
        {
            readRecord(record, bamFileIn);
            writeRecord(bamFileOut, record);
        }
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

```
@HD         VN:1.3         SO:coordinate
@SQ         SN:ref         LN:45
@SQ         SN:ref2         LN:40
r001         163         ref         7         30         8M4I4M1D3M         =         37         39         T
↪12561,2,20,112
r002         0         ref         9         30         1S2I6M1P1I1P1I4M2I         *         0         0
r003         0         ref         9         30         5H6M         *         0         0         AGCTAA
r004         0         ref         16         30         6M14N1I5M         *         0         0         ATAG
r003         16         ref         29         30         6H5M         *         0         0         TAGGC
r001         83         ref         37         30         9M         =         7         -
↪39         CAGCGCCAT         *
```

We instantiate a BamFileIn object for reading and a BamFileOut object for writing. First, we read the BAM header with readRecord and we write it with writeRecord. Then, we read each record from the input file and print it back on standard output. The alignment records are read into BamAlignmentRecord objects, which we will focus on below.

Assignment 1

Type Reproduction

Objective Create a file with the sample SAM content from above and adjust the path `"example.sam"` to the path to your SAM file (e.g. `"/path/to/my_example.sam"`).

Solution

```
#include <seqan/bam_io.h>

using namespace seqan;

int main()
{
    CharString bamFileName = getAbsolutePath("demos/tutorial/sam_and_bam_io/
↪example.sam");

    // Open input file, BamFileIn can read SAM and BAM files.
    BamFileIn bamFileIn;
    if (!open(bamFileIn, toCString(bamFileName)))
    {
        std::cerr << "ERROR: Could not open " << bamFileName << std::endl;
        return 1;
    }
    // Open output file, BamFileOut accepts also an ostream and a format tag.
    BamFileOut bamFileOut(context(bamFileIn), std::cout, Sam());

    try
    {
        // Copy header.
        BamHeader header;
        readHeader(header, bamFileIn);
        writeHeader(bamFileOut, header);

        // Copy records.
        BamAlignmentRecord record;
        while (!atEnd(bamFileIn))
        {
            readRecord(record, bamFileIn);
            writeRecord(bamFileOut, record);
        }
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Accessing the Header

Sequence information (i.e. @SQ records) from the BAM header is stored in the BamIOContext. All remaining BAM header information is stored in the class BamHeader.

Important: The header is not mandatory in SAM files and might be missing.

The following program accesses the BamIOContext of its BamFileIn and prints the reference sequence names and lengths present in the BAM header.

```
#include <seqan/bam_io.h>
```

```
using namespace seqan;

int main()
{
    CharString bamFileName = getAbsolutePath("demos/tutorial/sam_and_bam_io/example.
↪sam");

    BamFileIn bamFileIn(toCString(bamFileName));

    BamHeader header;
    readHeader(header, bamFileIn);

    typedef FormattedFileContext<BamFileIn, void>::Type TBamContext;

    TBamContext const & bamContext = context(bamFileIn);

    for (unsigned i = 0; i < length(contigNames(bamContext)); ++i)
        std::cout << contigNames(bamContext)[i] << '\t'
                  << contigLengths(bamContext)[i] << '\n';

    return 0;
}
```

The output looks like this:

```
ref        45
ref2        40
```

Accessing the Records

The class BamAlignmentRecord stores one alignment record of a SAM or BAM file. The class gives an in-memory representation that

1. is independent of whether it comes from/goes to a SAM or BAM file,
2. at the same time follows both formats closely,
3. allows for efficient storage and usage in C++ and
4. integrates well with the rest of the SeqAn library.

The following definition gives an overview of the available fields, their types, and how they map to the SAM and BAM fields. Note that we use the CigarElement class to store entries in the CIGAR string.

```
using namespace seqan;

class myBamAlignmentRecord
{
public:
    CharString qName;               // QNAME
    __uint16 flag;                  // FLAG
    int32_t rID;                    // REF
    int32_t beginPos;               // POS
    __uint8 mapQ;                   // MAPQ mapping quality, 255 for */invalid
    __uint16 bin;                   // bin for indexing
    String<CigarElement<> > cigar;  // CIGAR string
    int32_t rNextId;                // RNEXT (0-based)
    int32_t pNext;                  // PNEXT (0-based)
```

```
    int32_t tLen;                   // TLEN
    CharString seq;                 // SEQ, as in SAM/BAM file.
    CharString qual;                // Quality string as in SAM (Phred).
    CharString tags;                // Tags, raw as in BAM.

    // Constants for marking pos, reference id and length members invalid (== 0/*).
    static int32_t const INVALID_POS = -1;
    static int32_t const INVALID_REFID = -1;
    static int32_t const INVALID_LEN = 0;
};
```

The static members `INVALID_POS`, `INVALID_REFID`, and `INVALID_LEN` store sentinel values for marking positions, reference sequence ids, and lengths as invalid or N/A.

Tip: A BamAlignmentRecord is linked to a reference sequence by the field `rID`. The reference sequence information is stored in the BAM header and kept in the BamIOContext. To easily access reference sequence name and and length relative to a given BamAlignmentRecord within a BamFileIn, use functions getContigName and getContigLength.

An important related type is the enum BamFlags that provides constants for bit operations on the `flag` field. The functions hasFlagAllProper, hasFlagDuplicate, hasFlagFirst, hasFlagLast, hasFlagMultiple, hasFlagNextRC, hasFlagNextUnmapped, hasFlagQCNoPass, hasFlagRC, hasFlagSecondary, hasFlagUnmapped, and hasFlagSupplementary allow for easy reading of flags.

Assignment 2

Counting Records

Type Review

Objective Count the number of unmapped reads.

Hints Use the function hasFlagUnmapped.

Solution

```
#include <seqan/bam_io.h>

using namespace seqan;

int main()
{
    CharString bamFileName = getAbsolutePath("demos/tutorial/sam_and_bam_io/
↪example.sam");

    // Open input file.
    BamFileIn bamFileIn;
    if (!open(bamFileIn, toCString(bamFileName)))
    {
        std::cerr << "ERROR: Could not open " << bamFileName << std::endl;
        return 1;
    }

    unsigned numUnmappedReads = 0;

    try
    {
```

```
        // Read header.
        BamHeader header;
        readHeader(header, bamFileIn);

        // Read records.
        BamAlignmentRecord record;
        while (!atEnd(bamFileIn))
        {
            readRecord(record, bamFileIn);
            if (hasFlagUnmapped(record))
                numUnmappedReads += 1;
        }
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    std::cout << "Number of unmapped reads: " << numUnmappedReads << "\n";

    return 0;
}
```

```
Number of unmapped reads: 0
```

Accessing the Records' Tags

You can use the BamTagsDict class to access the the tag list of a record in a dictionary-like fashion. This class also performs the necessary casting when reading and writing tag list entries.

BamTagsDict acts as a wrapper around the raw `tags` member of a BamAlignmentRecord, which is of type CharString:

```
using namespace seqan;

    BamAlignmentRecord record;
    BamTagsDict tagsDict(record.tags);
```

We can add a tag using the function setTagValue. When setting an already existing tag's value, its value will be overwritten. Note that in the following, we give the tags value in SAM format because it is easier to read, although they are stored in BAM format internally.

```
    setTagValue(tagsDict, "NM", 2);
    // => tags: "NM:i:2"
    setTagValue(tagsDict, "NH", 1);
    // => tags: "NM:i:2 NH:i:1"
    setTagValue(tagsDict, "NM", 3);
    // => tags: "NM:i:3 NH:i:1"
```

The first parameter to setTagValue is the BamTagsDict, the second one is a two-character string with the key, and the third one is the value. Note that the type of tag entry will be taken automatically from the type of the third parameter.

Reading values is slightly more complex because we have to handle the case that the value is not present. First, we get the index of the tag in the tag list.

```
unsigned tagIdx = 0;
if (!findTagKey(tagIdx, tagsDict, "NH"))
    std::cerr << "ERROR: Unknown key!\n";
```

Then, we can read the value from the BamTagsDict using the function extractTagValue.

```
int tagValInt = 0;
if (!extractTagValue(tagValInt, tagsDict, tagIdx))
    std::cerr << "ERROR: There was an error extracting NH from tags!\n";
```

The function returns a `bool` that is `true` on success and `false` otherwise. The extraction can fail if the index is out of bounds or the value in the dictionary cannot be cast to the type of the first parameter.

The value in the tags dictionary will be casted to the type of the first parameter of extractTagValue:

```
short tagValShort = 0;
extractTagValue(tagValShort, tagsDict, tagIdx);
```

Assignment 3

Reading Tags

Type Review

Objective Modify the solution of Assignment 2 to count the number of records having the `"XX"` tag.

Solution

```
#include <seqan/bam_io.h>

using namespace seqan;

int main()
{
    CharString bamFileName = getAbsolutePath("demos/tutorial/sam_and_bam_io/
↪example.sam");

    // Open input file.
    BamFileIn bamFileIn;
    if (!open(bamFileIn, toCString(bamFileName)))
    {
        std::cerr << "ERROR: Could not open " << bamFileName << std::endl;
        return 1;
    }

    unsigned numXXtags = 0;

    try
    {
        // Read header.
        BamHeader header;
        readHeader(header, bamFileIn);

        // Rear records.
        BamAlignmentRecord record;
        while (!atEnd(bamFileIn))
        {
```

```
            readRecord(record, bamFileIn);
            BamTagsDict tagsDict(record.tags);

            unsigned tagIdx = 0;
            if (findTagKey(tagIdx, tagsDict, "XX"))
                numXXtags += 1;
        }
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    std::cout << "Number of records with the XX tag: " << numXXtags << "\n";

    return 0;
}
```

```
Number of records with the XX tag: 1
```

Using BAM Indices

SeqAn also contains features for reading BAM indices with the format `.bai`. These indices can be built using the `samtools index` command. In the near future we plan to support building the bam index with SeqAn as well.

You can read indices into a BaiBamIndex object with the function open. Then, you can use the function jumpToRegion to jump to a specific position within BAM files. After jumping, the next record to be read is before the given region. Therefore, you have to skip records until you access the one you are looking for.

```
#include <seqan/sequence.h>
#include <seqan/bam_io.h>

using namespace seqan;

int main(int argc, char const * argv[])
{
    CharString bamFileName = getAbsolutePath("demos/tutorial/sam_and_bam_io/example.
↪bam");
    CharString baiFileName = getAbsolutePath("demos/tutorial/sam_and_bam_io/example.
↪bam.bai");
    CharString rName = "ref";

    // Open BamFileIn for reading.
    BamFileIn inFile;
    if (!open(inFile, toCString(bamFileName)))
    {
        std::cerr << "ERROR: Could not open " << bamFileName << " for reading.\n";
        return 1;
    }

    // Read BAI index.
    BamIndex<Bai> baiIndex;
    if (!open(baiIndex, toCString(baiFileName)))
    {
        std::cerr << "ERROR: Could not read BAI index file " << baiFileName << "\n";
```

```
        return 1;
    }

    // Read header.
    BamHeader header;
    readHeader(header, inFile);

    // Translate from reference name to rID.
    int rID = 0;
    if (!getIdByName(rID, contigNamesCache(context(inFile)), rName))
    {
        std::cerr << "ERROR: Reference sequence named " << rName << " not known.\n";
        return 1;
    }

    // Translate BEGIN and END arguments to number, 1-based to 0-based.
    int beginPos = 9, endPos = 30;

    // 1-based to 0-based.
    beginPos -= 1;
    endPos -= 1;

    // Translate number of elements to print to number.
    int num = 3;

    // Jump the BGZF stream to this position.
    bool hasAlignments = false;
    if (!jumpToRegion(inFile, hasAlignments, rID, beginPos, endPos, baiIndex))
    {
        std::cerr << "ERROR: Could not jump to " << beginPos << ":" << endPos << "\n";
        return 1;
    }
    if (!hasAlignments)
        return 0;  // No alignments here.

    // Seek linearly to the selected position.
    BamAlignmentRecord record;
    int numPrinted = 0;
    BamFileOut out(inFile, std::cout, Sam());

    while (!atEnd(inFile) && numPrinted < num)
    {
        readRecord(record, inFile);

        // If we are on the next reference or at the end already then we stop.
        if (record.rID == -1 || record.rID > rID || record.beginPos >= endPos)
            break;
        // If we are left of the selected position then we skip this record.
        if (record.beginPos < beginPos)
            continue;

        // Otherwise, we print it to the user.
        numPrinted++;
        writeRecord(out, record);
    }

    return 0;
}
```

```
r002        0        ref        9        30        1S2I6M1P1I1P1I4M2I        *        0        0
↪
r003        0        ref        9        30        5H6M        *        0        0        AGCTAA
↪
r004        0        ref        16        30        6M14N1I5M        *        0        0        ATAG
↪
```

Next Steps

- Read the SAM Format Specification.
- Continue with the *Tutorials*.

ToC

Contents

VCF I/O

Learning Objective In this tutorial, you will learn how to read and write VCF files.

Difficulty Average

Duration 1 h (45 min if you know the VCF format)

Prerequisites *Sequences*, *File I/O Overview*, VCF Format Specification (v4.2)

Overview

This tutorial shows how to read and write VCF files using the VcfFileIn and VcfFileOut classes. It starts out with a quick reminder on the structure of VCF files and will then continue with how to read and write VCF files and access the tags of a record.

Important: Note that this tutorial is targeted at readers that already know about the VCF format. If you do not know about the VCF format yet, then this tutorial will be harder for your to understand.

The VCF format allows storing genomic variants of individuals with respect to a reference. The general file structure starts with

1. several meta-information lines starting with ##,
2. one header line giving the names of the individuals, and
3. an arbitrary number of records.

The information of (1) and (2) will be read and written together as the "header" of the file. For simple variants such as SNPs and small indels, each record corresponds to a variant. More complex variants can be stored in multiple records (see the VCF standard on "breakends" for more information).

The `vcf_io` module of SeqAn allows to read and write VCF files record-wise. Since the structure of the fields in the VCF format often is very complex and the format undergoes changes in this respect, SeqAn only offers basic parsing functionality: The position is stored as a 0-based integer, reference names are stored in a reference name store (similar as in the *SAM and BAM I/O* Tutorial), and the quality is stored as a `float` value.

The remaining fields have to be parsed from and composed as strings in the user's application.

VCF Format

This section gives a very brief overview of the VCF file structure. For more details, see the VCF Format Specification (v4.2).

The following is an example of a VCF file:

```
##fileformat=VCFv4.1
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
##contig=<ID=20,length=62435964,assembly=B36,md5=f126cdf8a6e0c7f379d618ff66beb2da,
↪species="Homo sapiens",taxonomy=x>
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">

↪#CHROM        POS         ID        REF         ALT        QUAL        FILTER        INFO        FORM
20        14370        rs6054257        G        A        29        PASS        NS=3;
↪DP=14;AF=0.5;DB;H2        GT:GQ:DP:HQ        0|0:48:1:51,51        1|0:48:8:51,
↪51        1/1:43:5:.,.
20        17330        .        T        A        3        q10        NS=3;DP=11;AF=0.
↪017        GT:GQ:DP:HQ        0|0:49:3:58,50        0|1:3:5:65,3        0/0:41:3
20        1110696        rs6040355        A        G,
↪T        67        PASS        NS=2;DP=10;AF=0.333,0.667;AA=T;
↪DB        GT:GQ:DP:HQ        1|2:21:6:23,27        2|1:2:0:18,2        2/2:35:4
20        1230237        .        T        .        47        PASS        NS=3;DP=13;
↪AA=T        GT:GQ:DP:HQ        0|0:54:7:56,60        0|0:48:4:51,51        0/0:61:2
20        1234567        microsat1        GTC        G,
↪GTCT        50        PASS        NS=3;DP=9;AA=G        GT:GQ:DP        0/
↪1:35:4        0/2:17:2        1/1:40:3
```

The file starts with meta information lines (starting with ##) with a key/value structure. The most important lines have the keys **contig**, **INFO**, **FILTER**, and **FORMAT**.

contig Lines with this key list the contigs of the reference genome.‘‘

INFO These lines give valid keys (and the format of the values) for the INFO column.

FILTER Valid values of the FILTER column.

FORMAT Valid entries for the INFO column.

The meta information lines are followed by the header line which gives the names of the first 9 columns which are always the same (CHROM, POS, ID, REF, ALT, QUAL, FILTER, INFO, FORMAT) and a non-empty list of sample names. The columns are separated by spaces.

The header line is followed by the records which contains a value for each column in the header.

CHROM Name of the chromosome/reference sequence that the variant lies on.

POS The 1-based position of the variant.

ID A name of the variant. `.` is used if no name is available.

REF The value of the reference allele.

ALT The alternate allele values (multiple values are comma-separated).

QUAL Quality value of the call (float).

FILTER A value for the filter result (given in a `FILTER` meta information line).

INFO Information about a variant.

FORMAT Colon-separated list of entries that are found for each variant.

The 9 mandatory columns are followed by as many columns as there are individuals. For each individual, there is a colon-separated list of values in the order given in the FORMAT cell.

Tip: 1-based and 0-based positions.

There are two common ways of specifying intervals.

1. Start counting positions at 1 and give intervals by the first and last position that are part of the interval (closed intervals). For example, the interval `[1000; 2000]` starts at character 1000 and ends at character 2000 and includes it. This way is natural to non-programmers and used when giving coordinates in GFF files or genome browsers such as UCSC Genome Browser and IGV.
2. Start counting positions at 0 and give intervals by the first position that is part of the interval and giving the position behind the last position that is part of the interval. The interval from above would be `[999; 2000)` in this case.

In text representations, such as VCF, 1-based closed intervals are used whereas in the internal binary data structures, SeqAn uses 0-based half-open intervals. When fields are read as text, numbers are not translated, of course.

A First Working Example

The following example shows a program that reads the file `example.vcf` and prints its contents back to the user on standard output.

```
#include <seqan/vcf_io.h>

using namespace seqan;

int main()
{
    // Open input file.
    VcfFileIn vcfIn(toCString(getAbsolutePath("demos/tutorial/vcf_io/example.vcf")));

    // Attach to standard output.
    VcfFileOut vcfOut(vcfIn);
    open(vcfOut, std::cout, Vcf());

    // Copy over header.
    VcfHeader header;
    readHeader(header, vcfIn);
    writeHeader(vcfOut, header);

    // Copy the file record by record.
    VcfRecord record;
    while (!atEnd(vcfIn))
    {
        readRecord(record, vcfIn);
        writeRecord(vcfOut, record);
    }

    return 0;
}
```

The program first opens a VcfFileIn for reading the file, then a VcfFileOut for writing it. First, the header is copied by means of a VcfHeader object that we will see below. Then, the input file is read record by record and written out to standard output. The alignment records are read into VcfRecord objects which we will focus on below.

The output of the example program looks as follows:

```
##fileformat=VCFv4.1
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
##contig=<ID=20,length=62435964,assembly=B36,md5=f126cdf8a6e0c7f379d618ff66beb2da,
 ↪species="Homo sapiens",taxonomy=x>
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AF,Number=A,Type=Float,Description="Allele Frequency">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=DB,Number=0,Type=Flag,Description="dbSNP membership, build 129">
##INFO=<ID=H2,Number=0,Type=Flag,Description="HapMap2 membership">
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">

 ↪#CHROM        POS         ID        REF         ALT         QUAL        FILTER        INFO        FORM
20        14370        rs6054257        G         A        29        PASS        NS=3;
 ↪DP=14;AF=0.5;DB;H2        GT:GQ:DP:HQ        0|0:48:1:51,51        1|0:48:8:51,
 ↪51        1/1:43:5:.,.
```

```
20        17330        .        T        A        3        q10        NS=3;DP=11;AF=0.
↪017        GT:GQ:DP:HQ        0|0:49:3:58,50        0|1:3:5:65,3        0/0:41:3
20        1110696        rs6040355        A        G,
↪T        67        PASS        NS=2;DP=10;AF=0.333,0.667;AA=T;
↪DB        GT:GQ:DP:HQ        1|2:21:6:23,27        2|1:2:0:18,2        2/2:35:4
20        1230237        .        T        .        47        PASS        NS=3;DP=13;
↪AA=T        GT:GQ:DP:HQ        0|0:54:7:56,60        0|0:48:4:51,51        0/0:61:2
20        1234567        microsat1        GTC        G,
↪GTCT        50        PASS        NS=3;DP=9;AA=G        GT:GQ:DP        0/
↪1:35:4        0/2:17:2        1/1:40:3
```

Assignment 1

Type Reproduction

Objective Create a file with the sample VCF content from above and adjust the path `"example.vcf"` to the path to your SAM file (e.g. `"/path/to/my_example.sam"`).

Solution

```
#include <seqan/vcf_io.h>

using namespace seqan;

int main()
{
    // Open input file.
    VcfFileIn vcfIn(toCString(getAbsolutePath("demos/tutorial/vcf_io/example.vcf
↪")));

    // Attach to standard output.
    VcfFileOut vcfOut(vcfIn);
    open(vcfOut, std::cout, Vcf());

    // Copy over header.
    VcfHeader header;
    readHeader(header, vcfIn);
    writeHeader(vcfOut, header);

    // Copy the file record by record.
    VcfRecord record;
    while (!atEnd(vcfIn))
    {
        readRecord(record, vcfIn);
        writeRecord(vcfOut, record);
    }

    return 0;
}
```

Accessing the Header

Sequence information from the VCF header is stored in the VcfIOContext. The VcfIOContext of a VcfFileIn can be accessed through the function context. The VCF sequence informations can be in turn accessed through functions contigNames and sampleNames. All remaining VCF header information is stored in the class VcfHeader.

Accessing the Records

The class VcfRecord stores one record in a VCF file. It is best explained by its definition. Note how most fields are represented by Strings:

```
using namespace seqan;

class VcfRecord
{
public:
    int32_t rID;                          // CHROM
    int32_t beginPos;                     // POS
    CharString id;                        // ID
    CharString ref;                       // REF
    CharString alt;                       // ALT
    float qual;                           // QUAL
    CharString filter;                    // FILTER
    CharString info;                      // INFO
    CharString format;                    // FORMAT
    StringSet<CharString> genotypeInfos;  // <individual1> <individual2> ..

    // Constants for marking reference id and position as invalid.
    static const int32_t INVALID_REFID = -1;
    static const int32_t INVALID_POS = -1;
    // This function returns the float value for "invalid quality".
    static float MISSING_QUAL();
};
```

The static members `INVALID_POS` and `INVALID_REFID` store sentinel values for marking positions and reference sequence ids as invalid. The static funtion `MISSING_QUAL()` returns the IEEE float "NaN" value.

Tip: A VcfRecord is linked to a reference sequence by the field `rID` and to samples by `genotypeInfos`. The sequence information is stored in the VCF header and kept in the VcfIOContext.

Assignment 2

Counting Records

Type Review

Objective Change the result of *Assignment 1* by counting the number of variants for each chromosome/contig instead of writing out the records.

Hints The reference sequence information from the VCF header is stored inside the VcfIOContext of its VcfFileIn. You can obtain the number of contigs from the length of the contigNames.

Solution

```
#include <seqan/vcf_io.h>

using namespace seqan;

int main()
{
    try
    {
```

```
        // Open input file.
        VcfFileIn vcfIn(toCString(getAbsolutePath("demos/tutorial/vcf_io/example.
↪vcf")));

        // Copy over header.
        VcfHeader header;
        readHeader(header, vcfIn);

        // Get array of counters.
        String<unsigned> counters;
        unsigned contigsCount = length(contigNames(context(vcfIn)));
        resize(counters, contigsCount, 0);

        // Read the file record by record.
        VcfRecord record;
        while (!atEnd(vcfIn))
        {
            readRecord(record, vcfIn);

            // Register record with counters.
            counters[record.rID] += 1;
        }

        // Print result.
        std::cout << "VARIANTS ON CONTIGS\n";
        for (unsigned i = 0; i < contigsCount; ++i)
            std::cout << contigNames(context(vcfIn))[i] << '\t'
                      << counters[i] << '\n';
    }
    catch (seqan::Exception const & e)
    {
        std::cerr << "ERROR:" << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

The output is

```
VARIANTS ON CONTIGS
20        5
```

Creating a New File

Assignment 3

Generating VCF From Scratch

Type Application

Objective Write a program that manually creates the VCF file from above and than prints it back on standard output.

Hint

- use VcfHeaderRecord to create a header record that can be appended to the VcfHeader
- use `appendValue()` to append information to the VcfIOContext or the VcfHeader

- use the direct member access operator `.` if you want to access the information of a VcfRecord

Solution

```
#include <seqan/vcf_io.h>

using namespace seqan;

int main()
{
    // Open output file.
    VcfFileOut out(std::cout, Vcf());

    // Fill sequence names.
    appendValue(contigNames(context(out)), "20");

    // Fill sample names.
    appendValue(sampleNames(context(out)), "NA00001");
    appendValue(sampleNames(context(out)), "NA00002");
    appendValue(sampleNames(context(out)), "NA00002");

    // Fill and write out headers - This is somewhat tedious.
    VcfHeader header;
    appendValue(header, VcfHeaderRecord("fileformat", "VCFv4.1"));
    appendValue(header, VcfHeaderRecord("fileDate", "20090805"));
    appendValue(header, VcfHeaderRecord("source", "myImputationProgramV3.1"));
    appendValue(header, VcfHeaderRecord("reference", "file:///seq/references/
↪1000GenomesPilot-NCBI36.fasta"));
    appendValue(header, VcfHeaderRecord("contig", "<ID=20,length=62435964,
↪assembly=B36,md5=f126cdf8a6e0c7f379d618ff66beb2da,species=\"Homo sapiens\",
↪taxonomy=x>"));
    appendValue(header, VcfHeaderRecord("phasing", "partial"));
    appendValue(header, VcfHeaderRecord("INFO", "<ID=NS,Number=1,Type=Integer,
↪Description=\"Number of Samples With Data\">"));
    appendValue(header, VcfHeaderRecord("INFO", "<ID=DP,Number=1,Type=Integer,
↪Description=\"Total Depth\">"));
    appendValue(header, VcfHeaderRecord("INFO", "<ID=AF,Number=A,Type=Float,
↪Description=\"Allele Frequency\">"));
    appendValue(header, VcfHeaderRecord("INFO", "<ID=AA,Number=1,Type=String,
↪Description=\"Ancestral Allele\">"));
    appendValue(header, VcfHeaderRecord("INFO", "<ID=DB,Number=0,Type=Flag,
↪Description=\"dbSNP membership, build 129\">"));
    appendValue(header, VcfHeaderRecord("INFO", "<ID=H2,Number=0,Type=Flag,
↪Description=\"HapMap2 membership\">"));
    appendValue(header, VcfHeaderRecord("FILTER", "<ID=q10,Description=\"Quality␣
↪below 10\">"));
    appendValue(header, VcfHeaderRecord("FILTER", "<ID=s50,Description=\"Less␣
↪than 50% of samples have data\">"));
    appendValue(header, VcfHeaderRecord("ID", "<ID=GT,Number=1,Type=String,
↪Description=\"Genotype\">"));
    appendValue(header, VcfHeaderRecord("ID", "<ID=GQ,Number=1,Type=Integer,
↪Description=\"Genotype Quality\">"));
    appendValue(header, VcfHeaderRecord("ID", "<ID=DP,Number=1,Type=Integer,
↪Description=\"Read Depth\">"));
    appendValue(header, VcfHeaderRecord("ID", "<ID=HQ,Number=2,Type=Integer,
↪Description=\"Haplotype Quality\">"));
    writeHeader(out, header);

    // Fill and write out the record.
```

```
    VcfRecord record;
    record.rID = 0;
    record.beginPos = 14369;
    record.id = "rs6054257";
    record.ref = "G";
    record.alt = "A";
    record.qual = 29;
    record.filter = "PASS";
    record.info = "NS=3;DP=14;AF=0.5;DB;H2";
    record.format = "GT:GQ:DP:HQ";
    appendValue(record.genotypeInfos, "0|0:48:1:51,51");
    appendValue(record.genotypeInfos, "1|0:48:8:51,51");
    appendValue(record.genotypeInfos, "1/1:43:5:.,.");
    writeRecord(out, record);

    return 0;
}
```

Next Steps

- Continue with the *Tutorials*

ToC

Contents

BED I/O

Learning Objective In this tutorial, you will learn how to read and write BED files.

Difficulty Average

Duration 45min

Prerequisites *Sequences*, *File I/O Overview*, BED Format Specification

This tutorial shows how to read and write BED files using the BedFileIn and BedFileOut classes. It starts out with a quick reminder on the structure of BED files and will then continue with how to read and write BED files.

Originally, the BED format was designed for storing annotation tracks in the UCSC genome browser. Such an annotation track consists of multiple annotation records. Each annotation adds some meta information to a genomic interval (an interval with begin/end position on a contig/chromosome) The original specification of the format can be found in the UCSC Genome Browser FAQ.

The BED format is a TSV format and contains 12 columns. The first three columns specify a genomic region (contig/chromsome name, begin, and end position) and the remaining columns contain additional information. The full format will be described below.

Since genomic intervals are very useful and because there were many tools for manipulating BED files (sorting, intersecting intervals etc.), many other authors and projects created variants of the BED format. Usually, three or more columns have the same meaning as in BED and are followed by other, arbitrary tab-separated columns with additional annotation information. The "full" BED format is then called BED12. BED3, BED4, BED5, and BED6 use the first 3-6 columns and keep the remaining information as data.

BED files can be manipuluated using standard Unix tools such as `sed`, `awk`, and `sort`. There also is the bedtools suite with additional functionality.

The SeqAn module `bed_io` allows the reading and writing of BED files.

BED Format

The following is an example of a BED file:

```
chr1        66999824        67210768        NM_
↪032291        0        +        6700004167208778        0        25        227,
↪64,25,72,57,55,176,12,12,25,52,86,93,75,501,128,127,60,112,156,133,203,65,
↪165,2013,        0,91705,98928,101802,105635,108668,109402,126371,133388,
↪136853,137802,139139,142862,145536,147727,155006,156048,161292,185152,
↪195122,199606,205193,206516,207130,208931,
chr1        48998526        50489626        NM_032785        0        -
↪        4899984450489468        0        14        1439,27,97,163,153,112,
↪115,90,40,217,95,125,123,192,        0,2035,6787,54149,57978,101638,120482,
↪130297,334336,512729,712915,1164458,1318541,1490908,
chr1        16767166        16786584        NM_
↪018090        0        +        1676725616785385        0        8        182,
↪101,105,82,109,178,76,1248,        0,2960,7198,7388,8421,11166,15146,18170,
chr1        33546713        33585995        NM_
↪052998        0        +        3354785033585783        0        12        182,
↪121,212,177,174,173,135,166,163,113,215,351,0,275,488,1065,2841,10937,
↪12169,13435,15594,16954,36789,38931,
chr1        16767166        16786584        NM_
↪001145278        0        +        1676725616785385        0        8        104,
↪101,105,82,109,178,76,1248,        0,2960,7198,7388,8421,11166,15146,18170,
```

The meaning of the columns are as follows:

ref (1) Name of the reference sequence.

beginPos (2) Begin position of the interval.

endPos (3) End position of the interval.

name (4) Name of the interval.

score (5) A score, could also be in scientific notation or several values in a comma/colon-separated list.

strand (6) The strand of the feature, + for forward, − for reverse, `.` for unknown/dont-care.

thickBegin (7) Begin position where the feature is drawn thick in the UCSC browser.

thickEnd (8) End position where the feature is drawn thick in the UCSC browser.

itemRgb (9) Comma-separated triple with RGB values (0..255 each)

blockCount (10) The number of blocks (exons) in the BED line (for the UCSC browser).

blockStarts (11) Comma-separated list with begin positions of exons (for the UCSC browser, should be consistent with `blockCount`).

blockSizes (12) Comma-separated list with exon lists (for the UCSC browser, should be consistent with `blockCount`).

Tip: 1-based and 0-based positions.

There are two common ways of specifying intervals.

1. Start counting positions at 1 and give intervals by the first and last position that are part of the interval (closed intervals). For example, the interval `[1000; 2000]` starts at character 1000 and ends at character 2000 and includes it. This way is natural to non-programmers and used when giving coordinates in GFF files or genome browsers such as UCSC Genome Browser and IGV.
2. Start counting positions at 0 and give intervals by the first position that is part of the interval and giving the position behind the last position that is part of the interval. The interval from above would be `[999; 2000)` in this case.

In text representations, such as GFF and GTF, 1-based closed intervals are used whereas in the internal binary data structures, SeqAn uses 0-based half-open intervals. BED is a text format using 0-based positions.

A First Working Example

The following example shows an example of a program that reads the file `example.bed` and prints its contents back to the user on standard output.

```
#include <seqan/bed_io.h>
using namespace seqan;

int main()
{
    // Open input bed file.
    BedFileIn bedIn(toCString(getAbsolutePath("demos/tutorial/bed_io/example.bed")));

    // Attach to standard output.
    BedFileOut bedOut(std::cout, Bed());

    // Copy the file record by record.
    BedRecord<Bed3> record;

    while (!atEnd(bedIn))
    {
        readRecord(record, bedIn);
        writeRecord(bedOut, record);
    }

    return 0;
}
```

The program first opens a BedFileIn for reading and a BedFileOut for writing. The BED records are read into BedRecord objects which we will focus on below. In this case, we use the Bed3Record specialization of the BedRecord class.

The output of the example program looks as follows:

```
chr1        66999824        67210768        NM_
↪032291        0        +        6700004167208778        0        25        227,64,
↪25,72,57,55,176,12,12,25,52,86,93,75,501,128,127,60,112,156,133,203,65,165,2013,
↪        0,91705,98928,101802,105635,108668,109402,126371,133388,136853,137802,
↪139139,142862,145536,147727,155006,156048,161292,185152,195122,199606,205193,206516,
↪207130,208931,
chr1        48998526        50489626        NM_032785        0        -
↪        4899984450489468        0        14        1439,27,97,163,153,112,115,90,40,
↪217,95,125,123,192,        0,2035,6787,54149,57978,101638,120482,130297,334336,
↪512729,712915,1164458,1318541,1490908,
chr1        16767166        16786584        NM_
↪018090        0        +        1676725616785385        0        8        182,101,
↪105,82,109,178,76,1248,        0,2960,7198,7388,8421,11166,15146,18170,
chr1        33546713        33585995        NM_
↪052998        0        +        3354785033585783        0        12        182,121,
↪212,177,174,173,135,166,163,113,215,351,0,275,488,1065,2841,10937,12169,13435,15594,
↪16954,36789,38931,
chr1        16767166        16786584        NM_
↪001145278        0        +        1676725616785385        0        8        104,
↪101,105,82,109,178,76,1248,        0,2960,7198,7388,8421,11166,15146,18170,
```

Assignment 1

Type Reproduction

Objective Create a file with the sample BED content from above and adjust the path `"example.bed"` to the path to your BED file (e.g. `"/path/to/my_example.bed"`).

Solution

```
#include <seqan/bed_io.h>

using namespace seqan;

int main()
{
    // Open input bed file.
    BedFileIn bedIn;
    if (!open(bedIn, toCString(getAbsolutePath("demos/tutorial/bed_io/example.bed
↪"))))
    {
        std::cerr << "ERROR: Could not open example.bed" << std::endl;
        return 1;
    }
    // Attach to standard output.
    BedFileOut bedOut(std::cout, Bed());

    // Read the file record by record.
    BedRecord<Bed3> record;

    try
    {
        while (!atEnd(bedIn))
        {
            readRecord(record, bedIn);
            writeRecord(bedOut, record);
        }
```

```
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Accessing the Records

The class BedRecord stores one record in a BED file. Note that there are various specializations, each storing a different number of fields. We show the quasi-definition of BedRecord below. The other specializations have less fields.

```
#include <seqan/basic.h>
#include <seqan/bed_io.h>

using namespace seqan;

class BedRecord
{
public:
   CharString ref;       // reference name
   int32_t rID;          // index in sequenceNames of BedFile
   int32_t beginPos;     // begin position of the interval
   int32_t endPos;       // end position of the interval
   CharString name;      // name of the interval
   CharString score;     // score of the interval
   char strand;          // strand of the interval

   int32_t thickBegin;  // begin position for drawing thickly
   int32_t thickEnd;    // end position for drawing thickly
   BedRgb itemRgb;      // color for the item
   int32_t blockCount;  // number of blocks/exons
   String<int32_t> blockSizes;   // block sizes
   String<int32_t> blockBegins;  // block begin positions

   CharString data;    // any data not fitting into other members

   // Constants for marking reference id and position as invalid.
   static const int32_t INVALID_REFID = -1;
   static const int32_t INVALID_POS = -1;
};
```

The static members `INVALID_POS` and `INVALID_REFID` store sentinel values for marking positions and reference sequence ids as invalid.

Assignment 2

Counting Records

Type Review

Objective Change the result of *Assignment 1* by counting the number of variants for each chromosome/contig instead of writing out the records.

Solution

```
#include <seqan/bed_io.h>
#include <seqan/misc/name_store_cache.h>

using namespace seqan;

int main()
{
    // Open input bed file.
    BedFileIn bedIn;
    if (!open(bedIn, toCString(getAbsolutePath("demos/tutorial/bed_io/example.bed
↪"))))
    {
        std::cerr << "ERROR: Could not open example.bed\n";
        return 1;
    }

    // Array of counters and sequence names.
    String<unsigned> counters;
    StringSet<CharString> seqNames;
    NameStoreCache<StringSet<CharString> > cache(seqNames);

    // Read the file record by record.
    BedRecord<Bed3> record;

    try
    {
        while (!atEnd(bedIn))
        {
            readRecord(record, bedIn);
            unsigned rID = nameToId(cache, record.ref);

            // Resize counters if necessary and increment counter.
            assignValueById(counters, rID, getValueById(counters, rID) + 1);
        }
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    // Print result.
    std::cout << "RECORDS ON CONTIGS\n";
    for (unsigned i = 0; i < length(seqNames); ++i)
        if (counters[i] != 0u)
            std::cout << seqNames[i] << '\t' << counters[i] << '\n';

    return 0;
}
```

The output is

```
RECORDS ON CONTIGS
chr1        5
```

Creating a New File

Assignment 3

Generating BED From Scratch

Type Application

Objective Write a program that prints the following BED file. Create `BedRecord<Bed6>` objects and write them to a `BedFileOut` using `writeRecord()`.

```
chr7        127471195        127472363        Pos1        0        +
chr7        127472362        127473530        Pos2        0        +
```

Solution

```
#include <seqan/bed_io.h>

#include <sstream>

using namespace seqan;

int main()
{
    BedFileOut out(std::cout, Bed());

    BedRecord<Bed6> record;

    // Fill and write out the first record.
    record.ref = "chr7";
    record.beginPos = 127471195;
    record.endPos = 127472363;
    record.name = "Pos1";
    record.score = "0";
    record.strand = '+';
    writeRecord(out, record);

    // Fill and write out the second record.
    record.ref = "chr7";
    record.beginPos = 127472362;
    record.endPos = 127473530;
    record.name = "Pos2";
    record.score = "0";
    record.strand = '+';
    writeRecord(out, record);

    return 0;
}
```

Next Steps

- Continue with the *Tutorials*.

ToC

Contents

GFF and GTF I/O

Learning Objective In this tutorial, you will learn how to read and write GFF and GTF files.

Difficulty Average

Duration 45 min

Prerequisites *Sequences*, *File I/O Overview*, GFF Format Specification

This tutorial shows how to read and write GFF and GTF files using the GffFileIn and GffFileOut classes. It starts out with a quick reminder on the structure of GFF and GTF files and will then continue with how to read and write GFF and GTF files.

The GFF and GTF formats are used for annotating genomic intervals (an interval with begin/end position on a contig/chromosome). GFF exists in versions 2 and 3 and GTF is sometimes called "GFF 2.5". There are specifications for GFF 2, GFF 3, and GTF available elsewhere. GFF and GTF are TSV-based formats and in general have the same structure. The main difference is the underlying system/ontology for the annotation but also smaller differences in the format.

In this tutorial, we will focus on the format GFF 3 since it is the most current one with most complete tool support. The information of this tutorial can easily be translated to the other two formats.

The SeqAn module `gff_io` allows the reading and writing of the GFF and GTF formats.

Tip: Format Version Support in SeqAn

GffFileIn allows to read GFF files in version 2 and 3 and GTF files. For writing, GffFileOut supports only GFF 3 and GTF.

GFF Format

The following is an example of a GFF 3 file:

```
ctg123        .        gene        1000        9000        .        +        .
↪        ID=gene00001;Name=EDEN
ctg123        .        TF_binding_site        1000        1012        .
↪        +        .        Parent=gene00001
ctg123        .        mRNA        1050        9000        .        +        .
↪        ID=mRNA00001;Parent=gene00001
ctg123        .        mRNA        1050        9000        .        +        .
↪        ID=mRNA00002;Parent=gene00001
```

```
ctg123          .          mRNA          1300          9000          .          +          .
↪          ID=mRNA00003;Parent=gene00001
ctg123          .          exon          1300          1500          .          +          .
↪          Parent=mRNA00003
ctg123          .          exon          1050          1500          .          +          .
↪          Parent=mRNA00001,mRNA00002
ctg123          .          exon          3000          3902          .          +          .
↪          Parent=mRNA00001,mRNA00003
ctg123          .          exon          5000          5500          .          +          .
↪          Parent=mRNA00001,mRNA00002,mRNA00003
ctg123          .          exon          7000          9000          .          +          .
↪          Parent=mRNA00001,mRNA00002,mRNA00003
ctg123          .          CDS          1201          1500          .
↪          +          0          ID=cds00001;Parent=mRNA00001
ctg123          .          CDS          3000          3902          .
↪          +          0          ID=cds00001;Parent=mRNA00001
ctg123          .          CDS          5000          5500          .
↪          +          0          ID=cds00001;Parent=mRNA00001
ctg123          .          CDS          7000          7600          .
↪          +          0          ID=cds00001;Parent=mRNA00001
ctg123          .          CDS          1201          1500          .
↪          +          0          ID=cds00002;Parent=mRNA00002
ctg123          .          CDS          5000          5500          .
↪          +          0          ID=cds00002;Parent=mRNA00002
ctg123          .          CDS          7000          7600          .
↪          +          0          ID=cds00002;Parent=mRNA00002
ctg123          .          CDS          3301          3902          .
↪          +          0          ID=cds00003;Parent=mRNA00003
ctg123          .          CDS          5000          5500          .
↪          +          1          ID=cds00003;Parent=mRNA00003
ctg123          .          CDS          7000          7600          .
↪          +          1          ID=cds00003;Parent=mRNA00003
ctg123          .          CDS          3391          3902          .
↪          +          0          ID=cds00004;Parent=mRNA00003
ctg123          .          CDS          5000          5500          .
↪          +          1          ID=cds00004;Parent=mRNA00003
ctg123          .          CDS          7000          7600          .
↪          +          1          ID=cds00004;Parent=mRNA00003
```

The meaning of the columns are as follows:

seq id (1) Name of the reference sequence.

source (2) Free text field describing the source of the annotation, such as a software (e.g. "Genescan") or a a database (e.g. "Genebank"), "`.`" for none.

type (3) The type of the annotation.

start (4) The 1-based begin position of the annotation.

end (5) The 1-based end position of the annotation.

score (6) The score of the annotation, "`.`" for none.

strand (7) The strand of the annotation, "+" and "−" for forward and reverse strand, "`.`" for features that are not stranded.

phase (8) Shift of the feature regarding to the reading frame, one of "`0`", "`1`", "`2`", and "`.`" for missing/dont-care.

attributes (9) A list of key/value attributes. For GFF 3, this is a list of `key=value` pairs, separated by semicolons (e.g. `ID=cds00003;Parent=mRNA00003`). For GTF and GFF 2, this is a list of tuples, separated by

semicolon. The first entry gives the key, the following entries are values. Strings are generally enclosed in quotes (e.g. `Target "HBA_HUMAN" 11 55 ; E_value 0.0003`)

Tip: 1-based and 0-based positions.

There are two common ways of specifying intervals.

1. Start counting positions at 1 and give intervals by the first and last position that are part of the interval (closed intervals). For example, the interval `[1000; 2000]` starts at character 1,000 and ends at character 2000 and includes it. This way is natural to non-programmers and used when giving coordinates in GFF files or genome browsers such as UCSC Genome Browser and IGV.
2. Start counting positions at 0 and give intervals by the first position that is part of the interval and giving the position behind the last position that is part of the interval. The interval from above would be `[999; 2000)` in this case.

In text representations, such as GFF and GTF, 1-based closed intervals are used whereas in the internal binary data structures, SeqAn uses 0-based half-open intervals.

A First Working Example

The following example shows an example of a program that reads the file `example.gff` and prints its contents back to the user on standard output.

```
#include <seqan/basic.h>
#include <seqan/gff_io.h>

using namespace seqan;

int main()
{
    // Get path to example file.
    CharString file = getAbsolutePath("demos/tutorial/gff_and_gtf_io/example.gff");

    // Open input file.
    GffFileIn gffIn(toCString(file));

    // Attach to standard output.
    GffFileOut gffOut(std::cout, Gff());

    // Copy the file record by record.
    GffRecord record;
    while (!atEnd(gffIn))
    {
        readRecord(record, gffIn);
        writeRecord(gffOut, record);
    }

    return 0;
}
```

The program first opens a GffFileIn for reading and a GffFileOut for writing. The GFF records are read into GffRecord objects which we will focus on below.

Assignment 1

Type Reproduction

Objective Create a file with the sample GFF content from above and adjust the path `"example.gff"` to the path to your GFF file (e.g. `"/path/to/my_example.gff"`).

Solution

```
#include <seqan/gff_io.h>

using namespace seqan;

int main()
{
    // Get path to example file.
    CharString file = getAbsolutePath("demos/tutorial/gff_and_gtf_io/example.gff
↪");

    // Open input file.
    GffFileIn gffIn;
    if (!open(gffIn, toCString(file)))
    {
        std::cerr << "ERROR: Could not open example.gff" << std::endl;
        return 1;
    }

    // Attach to standard output.
    GffFileOut gffOut(std::cout, Gff());

    // Copy the file record by record.
    GffRecord record;

    try
    {
        while (!atEnd(gffIn))
        {
            readRecord(record, gffIn);
            writeRecord(gffOut, record);
        }
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

```
ctg123        .        gene        1000        9000        .        +        .
↪        ID=gene00001;Name=EDEN
ctg123        .        TF_binding_site        1000        1012        .
↪        +        .        Parent=gene00001
ctg123        .        mRNA        1050        9000        .        +        .
↪        ID=mRNA00001;Parent=gene00001
ctg123        .        mRNA        1050        9000        .        +        .
↪        ID=mRNA00002;Parent=gene00001
```

```
ctg123      .       mRNA        1300        9000        .       +       .
↪       ID=mRNA00003;Parent=gene00001
ctg123      .       exon        1300        1500        .       +       .
↪       Parent=mRNA00003
ctg123      .       exon        1050        1500        .       +       .
↪       Parent=mRNA00001,mRNA00002
ctg123      .       exon        3000        3902        .       +       .
↪       Parent=mRNA00001,mRNA00003
ctg123      .       exon        5000        5500        .       +       .
↪       Parent=mRNA00001,mRNA00002,mRNA00003
ctg123      .       exon        7000        9000        .       +       .
↪       Parent=mRNA00001,mRNA00002,mRNA00003
ctg123      .       CDS         1201        1500        .
↪       +       0       ID=cds00001;Parent=mRNA00001
ctg123      .       CDS         3000        3902        .
↪       +       0       ID=cds00001;Parent=mRNA00001
ctg123      .       CDS         5000        5500        .
↪       +       0       ID=cds00001;Parent=mRNA00001
ctg123      .       CDS         7000        7600        .
↪       +       0       ID=cds00001;Parent=mRNA00001
ctg123      .       CDS         1201        1500        .
↪       +       0       ID=cds00002;Parent=mRNA00002
ctg123      .       CDS         5000        5500        .
↪       +       0       ID=cds00002;Parent=mRNA00002
ctg123      .       CDS         7000        7600        .
↪       +       0       ID=cds00002;Parent=mRNA00002
ctg123      .       CDS         3301        3902        .
↪       +       0       ID=cds00003;Parent=mRNA00003
ctg123      .       CDS         5000        5500        .
↪       +       1       ID=cds00003;Parent=mRNA00003
ctg123      .       CDS         7000        7600        .
↪       +       1       ID=cds00003;Parent=mRNA00003
ctg123      .       CDS         3391        3902        .
↪       +       0       ID=cds00004;Parent=mRNA00003
ctg123      .       CDS         5000        5500        .
↪       +       1       ID=cds00004;Parent=mRNA00003
ctg123      .       CDS         7000        7600        .
↪       +       1       ID=cds00004;Parent=mRNA00003
```

Accessing the Records

The class GffRecord stores one record in a Gff file.

```
using namespace seqan;

class GffRecord
{
public:
    CharString ref;      // reference name
    int32_t rID;         // index in sequenceNames of GffFile
    CharString source;   // source free text descriptor
    CharString type;     // type of the feature
    int32_t beginPos;    // begin position of the interval
    int32_t endPos;      // end position of the interval
    float score;         // score of the annotation
    char strand;         // the strand
    char phase;          // one of '0', '1', '2', and '.'
```

```
    // The key/value list, split into a list of keys and values.
    StringSet<CharString> tagNames;
    StringSet<CharString> tagValues;

    // Returns float value for an invalid score.
    static float INVALID_SCORE();

    // Constants for marking reference id and position as invalid.
    static const int32_t INVALID_IDX = -1;
    static const int32_t INVALID_POS = -1;
};
```

The static members `INVALID_POS` and `INVALID_REFID` store sentinel values for marking positions and reference sequence ids as invalid. The static funtion `INVALID_SCORE()` returns the IEEE float "NaN" value.

Assignment 2

Counting Records

Type Review

Objective Change the result of *Assignment 1* by counting the number of variants for each chromosome/contig instead of writing out the records.

Solution

```
#include <seqan/gff_io.h>
#include <seqan/misc/name_store_cache.h>

using namespace seqan;

int main()
{
    // Get path to example file.
    CharString file = getAbsolutePath("demos/tutorial/gff_and_gtf_io/example.gff
↪");

    // Open input file.
    GffFileIn gffIn;
    if (!open(gffIn, toCString(file)))
    {
        std::cerr << "ERROR: Could not open example.gff" << std::endl;
        return 1;
    }

    // Array of counters and sequence names.
    String<unsigned> counters;
    StringSet<CharString> seqNames;
    NameStoreCache<StringSet<CharString> > cache(seqNames);

    // Read the file record by record.
    GffRecord record;

    try
    {
        while (!atEnd(gffIn))
```

```
        {
            readRecord(record, gffIn);
            unsigned rID = nameToId(cache, record.ref);

            // Resize counters if necessary and increment counter.
            assignValueById(counters, rID, getValueById(counters, rID) + 1);
        }
    }
    catch (Exception const & e)
    {
        std::cout << "ERROR: " << e.what() << std::endl;
        return 1;
    }

    // Print result.
    std::cout << "RECORDS ON CONTIGS\n";
    for (unsigned i = 0; i < length(seqNames); ++i)
        if (counters[i] != 0u)
            std::cout << seqNames[i] << '\t' << counters[i] << '\n';

    return 0;
}
```

The output is

```
RECORDS ON CONTIGS
ctg123        23
```

Creating a New File

Assignment 3

Generating GFF From Scratch

Type Application

Objective Write a program that prints the following GFF file. Create `GffRecord` objects and write them to a `GffFileOut` using `writeRecord()`.

Solution

```
#include <seqan/gff_io.h>

using namespace seqan;

int main()
{
    GffFileOut out(std::cout, Gff());

    GffRecord record;

    // Fill and write out the first record.
    record.ref = "ctg123";
    record.source = "";
    record.type = "gene";
    record.beginPos = 999;
    record.endPos = 9000;
```

```
    record.strand = '+';
    record.score = GffRecord::INVALID_SCORE();
    appendValue(record.tagNames, "ID");
    appendValue(record.tagValues, "gene0001");
    appendValue(record.tagNames, "Name");
    appendValue(record.tagValues, "EDEN");
    writeRecord(out, record);

    // Clear the record.
    clear(record.tagNames);
    clear(record.tagValues);

    // Fill and write out the second record.
    record.ref = "ctg123";
    record.source = "";
    record.type = "TF_binding_site";
    record.beginPos = 999;
    record.endPos = 1012;
    record.strand = '+';
    record.score = GffRecord::INVALID_SCORE();
    appendValue(record.tagNames, "Parent");
    appendValue(record.tagValues, "gene0001");
    writeRecord(out, record);

    return 0;
}
```

Next Steps

- Continue with the *Tutorials*.

SeqAn supports many standard file formats used in bioinformatics applications. The *first tutorial* shows you the basic concepts and data structures for handling file I/O. So this would be a good starting point to learn more about it. Otherwise, feel free to look into the tutorials for your desired file format and start reading and writing your files.

How-Tos

Recipes

After reading the Tutorial we recommend to read some of the following recipes:

ToC

Contents

Accessing Index Fibres

Overview

Basically each index consists of a set of tables, called fibres. The set of available fibres of an index `Index<TText, TSpec>` depends on the index specialization `TSpec`.

Fibres	Description
EsaText	The original text the index should be based on. Can be either a sequence or a StringSet.
EsaSA	The suffix array stores the begin positions of all suffixes in lexicographical order.
EsaLcp	The lcp table stores at position i the length of the longest common prefix between suffix with rank i and rank i+1.
EsaChildtab	See *[AKO04]*
EsaBwt	The Burrows-Wheeler table stores at position i the character left of the suffix with rank i.
EsaRawText	Virtually concatenates all strings of the EsaText fibre.

WOTDIndexFibres	Description
WotdText	The original text the index should be based on.
WotdSA	The suffix array stores the begin positions of all suffixes in lexicographical order.
WotdDir	*[GKS03]*
WotdRawText	Virtually concatenates all strings of the WotdText fibre.

DfiIndex-Fibres	Description	Type
DfiText	The original text the index should be based on.	First template argument of the Index. Can be either a sequence or a StringSet.
DfiSA	The suffix array stores the begin positions of all suffixes in lexicographical order.	String over the SAValue type of the index.
DfiDir	See *[GKS03]*.	String over the Size type of the index.
DfiRaw-Text	Virtually concatenates all strings of the DfiText fibre.	ContainerConcept over the alphabet of the text.

QGramIn-dexFibres	Description	Type
QGram-Text	The original text the index should be based on.	First template argument of the Index. Can be either a sequence or a StringSet.
QGramShape	The q-gram Shape.	Specified by the first template argument of IndexQGram.
QGramSA	The suffix array stores the begin positions of all suffixes in lexicographical order.	String over the SAValue type of the index.
QGramDir	The directory maps q-gram hash values to start indices in the QGramSA fibre.	String over the Size type of the index.
QGram-Counts	Stores numbers of occurrences per sequence of each q-gram in pairs (seqNo,count), count>0.	String over Pair of the Size type of the index.
QGram-CountsDir	The counts directory maps q-gram hash values to start indices in the QGramCounts fibre.	String over the Size type of the index.
QGram-Buck-etMap	Used by the OpenAddressingQGramIndex index to store the hash value occupancy in the QGramDir fibre.	String over the Value type of the shape.
QGram-RawText	Virtually concatenates all strings of the QGramText fibre.	ContainerConcept over the alphabet of the text.

The first column in each table above contains the tags to select the corresponding fibre. Most of these tags are aliases for the same tag, e.g. `EsaSA`, `QGramSA`, ... are aliases for `FibreSA`. If you write an algorithm that is generic in the

type of index, you can use `FibreText` to specify the fibre that stores the index text.

Creation

In most cases you don't need to create the fibres of an index by hand. Most algorithms and data structures create them automatically, e.g. Finder or VSTreeIterator. If you want to specify a certain index construction algorithm, have to recreate a fibre or manually access a fibre you can recreate or create on-demand a fibre by indexCreate and indexRequire. If your algorithm should behave differently depending on the presence or absence of a fibre (and the fibre should then not be created), you can test for presence by indexSupplied.

Access

The type of each fibre can be determined by the metafunction Fibre. To access a fibre you can use the function getFibre whose return type is the result of Fibre. The second argument of both functions is a tag to select a specific fibre. See the first column in the tables above. One fibre in every index is the text to be indexed itself. This fibre can be assigned during the construction. For the ease of use, there exist shortcuts to access frequently used fibres:

Shortcut	Expands To ...
indexBucketMap(index)	getFibre(index, FibreBucketMap())
indexBwt(index)	getFibre(index, FibreBwt())
indexChildtab(index)	getFibre(index, FibreChildtab())
indexCounts(index)	getFibre(index, FibreCounts())
indexCountsDir(index)	getFibre(index, FibreCountsDir())
indexLcp(index)	getFibre(index, FibreLcp())
indexRawText(index)	getFibre(index, FibreRawText())
indexSA(index)	getFibre(index, FibreSA())
indexShape(index)	getFibre(index, FibreShape())
indexText(index)	getFibre(index, FibreText())

and to access a single value:

Shortcut	Expands To ...
bwtAt(pos, index)	indexBwt(index)[pos]
childAt(pos, index)	indexChildtab(index)[pos]
dirAt(pos, index)	indexDir(index)[pos]
lcpAt(pos, index)	indexLcp(index)[pos]
rawtextAt(pos, index)	indexRawText(index)[pos]
saAt(pos, index)	indexSA(index)[pos]
textAt(pos, index)	indexText(index)[pos]

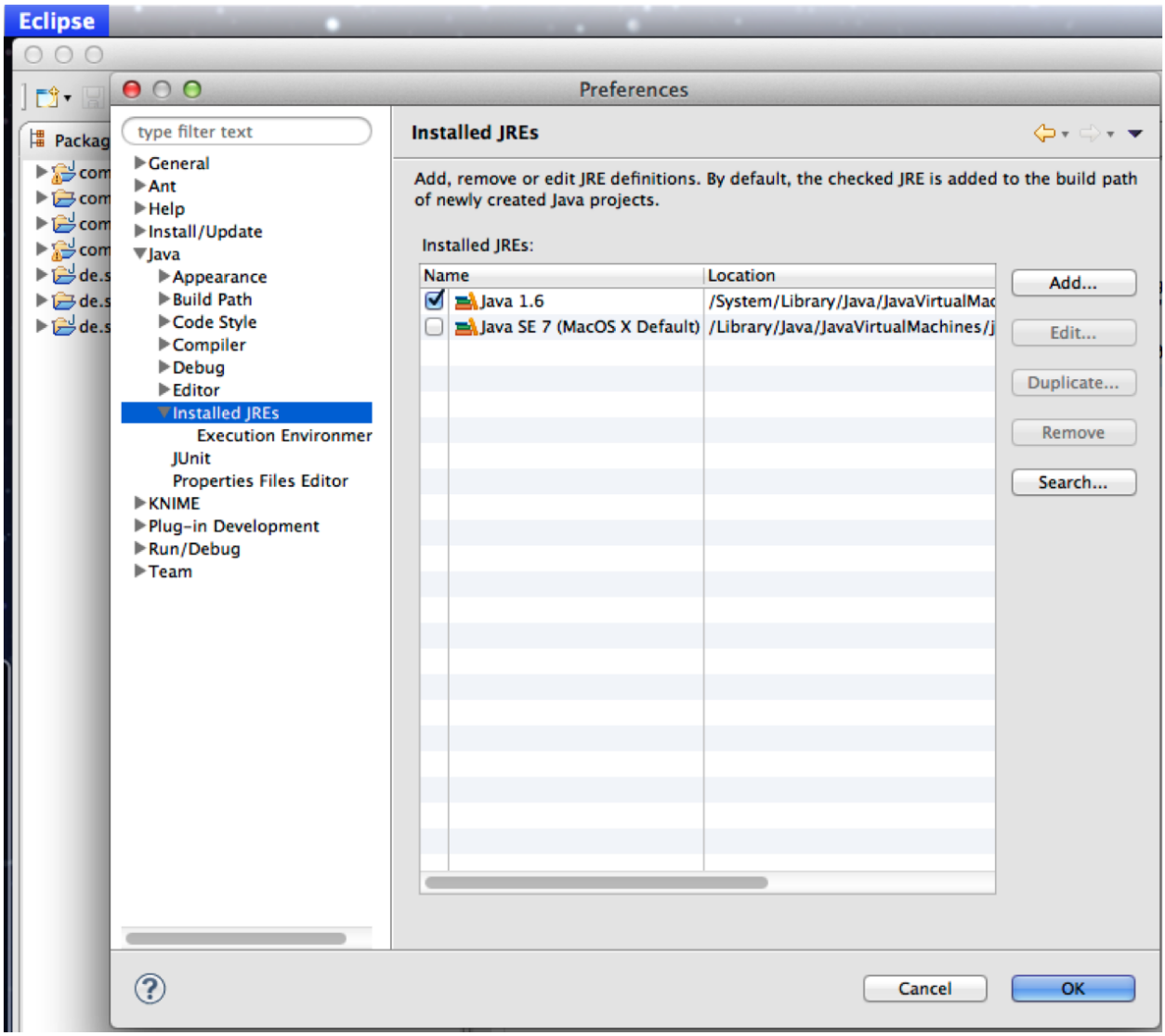
Please note that textAt can also be used if the index text is a StringSet. `pos` can then be a SAValue.

ToC

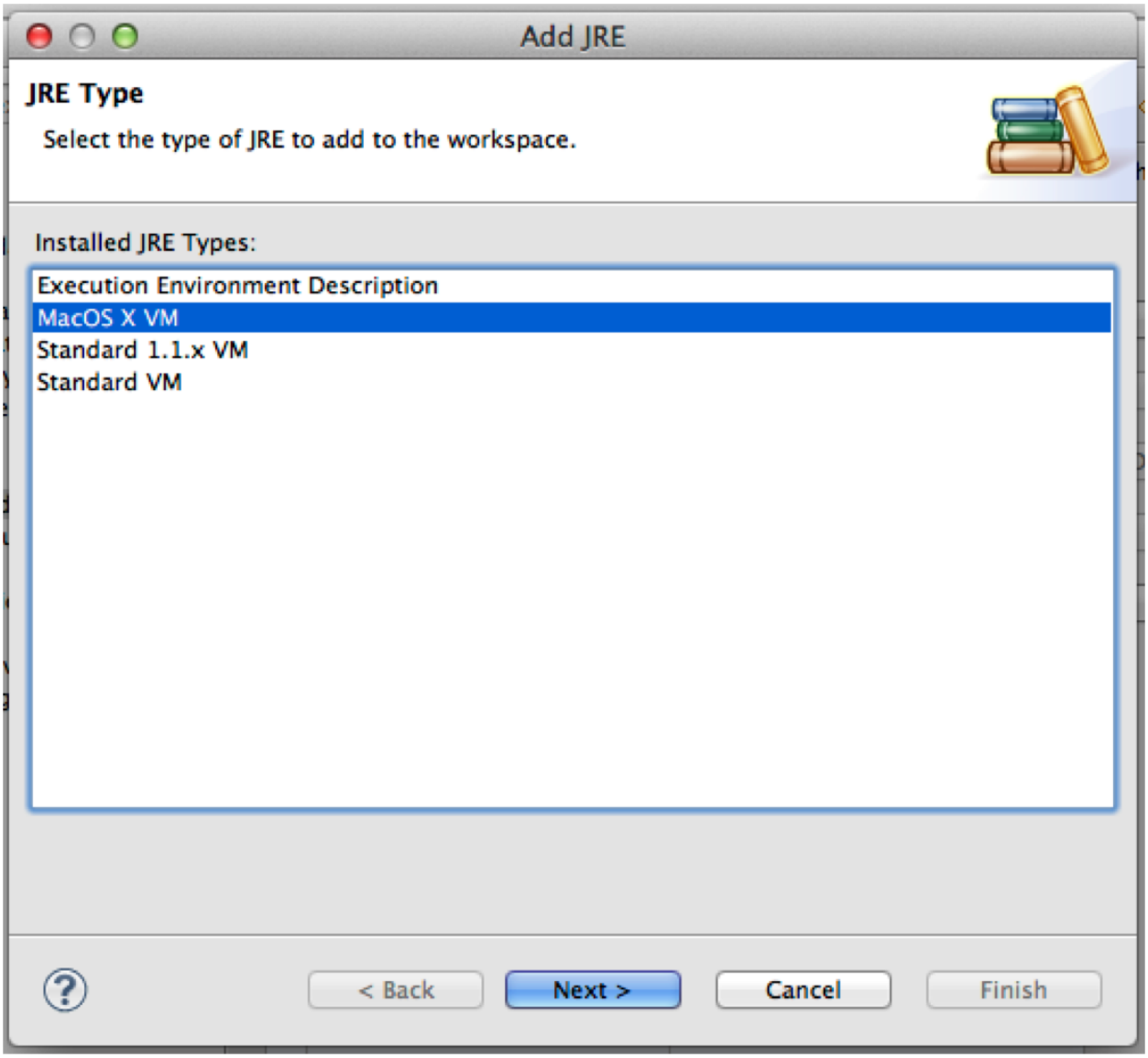
Contents

Choosing The JRE Version

In order to change the JRE to be used by KNIME go to Eclipse `Preferences` and select the `Java` menu.


Eclipse
Preferences
type filter text
General
Ant
Help
Install/Update
Java
Appearance
Build Path
Code Style
Compiler
Debug
Editor
Installed JREs
Execution Environmer
JUnit
Properties Files Editor
KNIME
Plug-in Development
Run/Debug
Team
Installed JREs
Add, remove or edit JRE definitions. By default, the checked JRE is added to the build path of newly created Java projects.
Installed JREs:
Name
Location
Java 1.6
/System/Library/Java/JavaVirtualMac
Java SE 7 (MacOS X Default)
/Library/Java/JavaVirtualMachines/j
Add...
Edit...
Duplicate...
Remove
Search...
Cancel
OK

Afterwards you can add the right JRE. Under MacOs you choose the entry `MacOS X VM`



then press next and select the right path, which should be `/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home` as shown here:

Press `Finish` and the right JRE will be added.

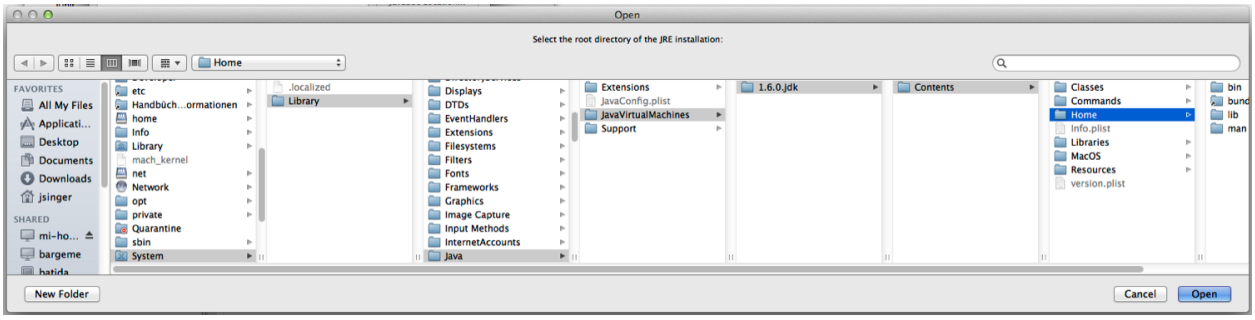
Afterwards you have to set the compiler options. In order to do so go to `Project` and select `Properties`.

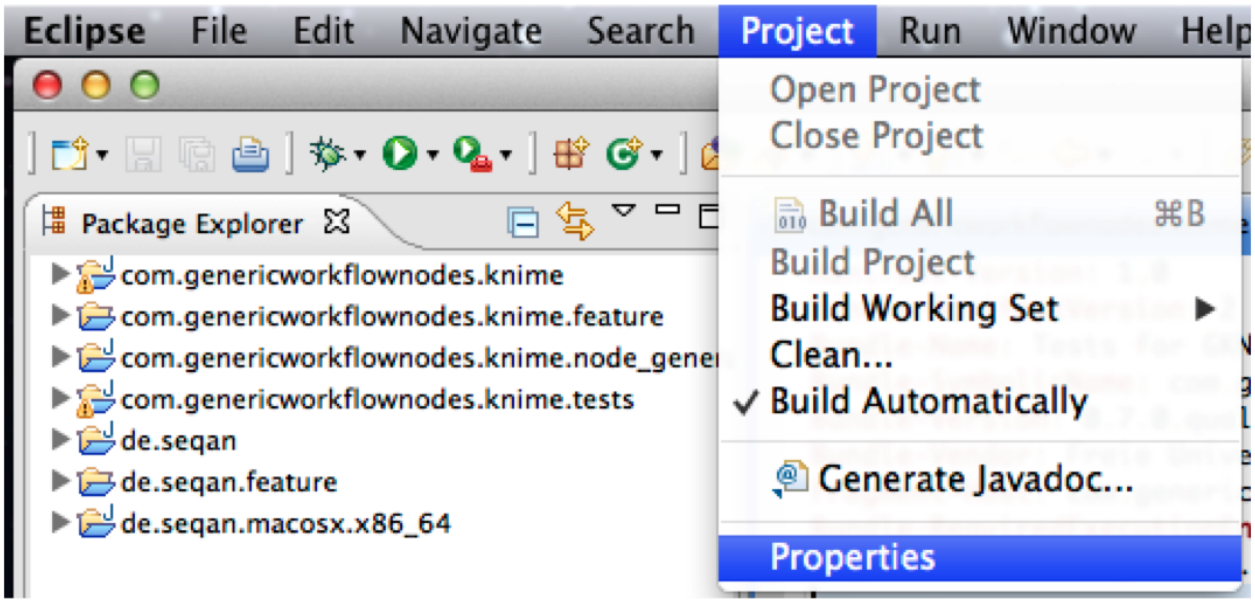
Now select `Java Compiler` and select the correct JRE at the `Compiler compliance level:`

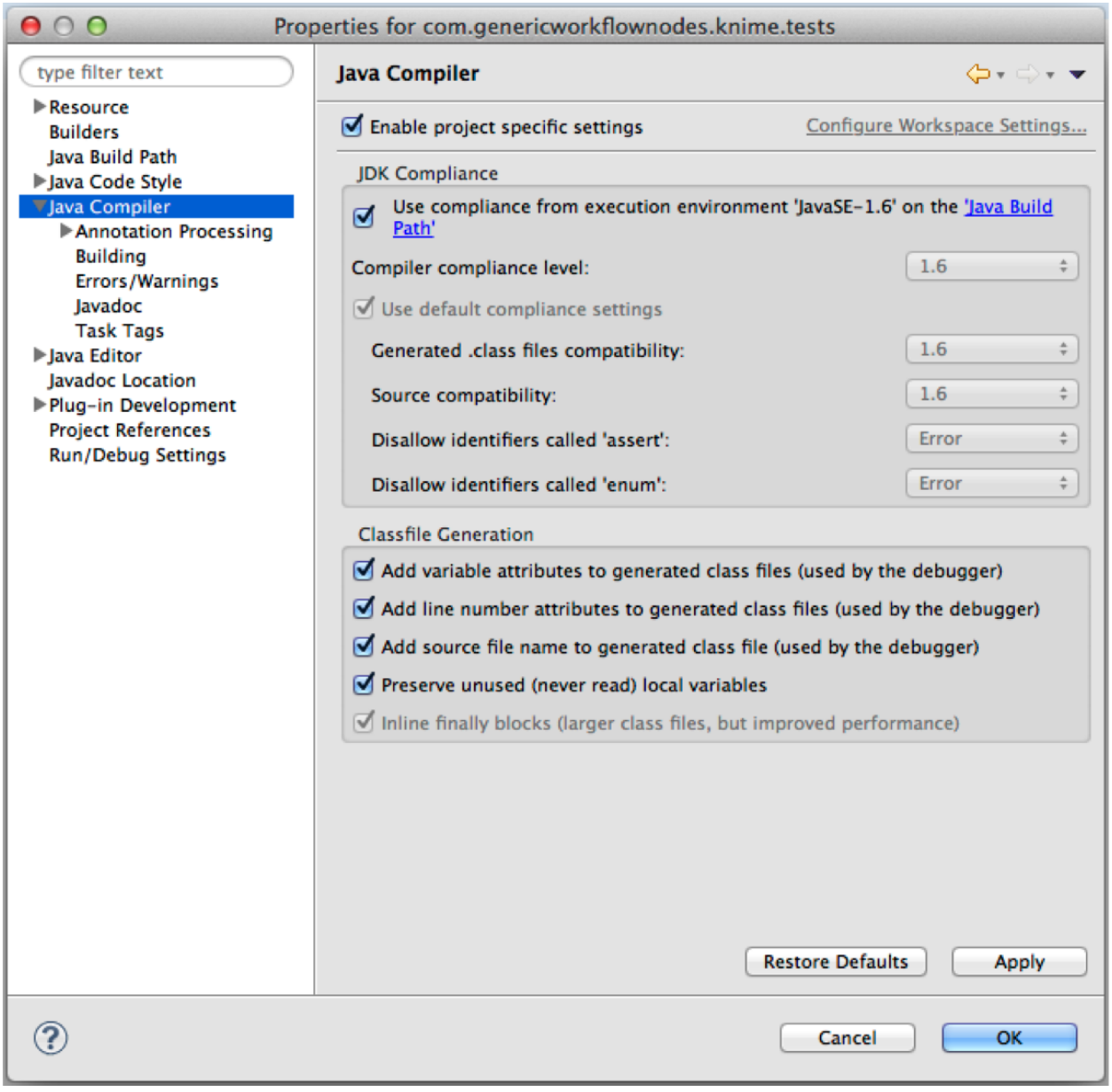
If you run the project now KNIME should start without problems.

ToC

Contents


Open
Select the root directory of the JRE installation:
Home
FAVORITES
All My Files
Applicati...
Desktop
Documents
Downloads
jsinger
SHARED
mi-ho...
bargeme
batida
etc
Handbüch...ormationen
home
Info
Library
mach_kernel
net
Network
opt
private
Quarantine
sbin
System
.localized
Library
Displays
DTDs
EventHandlers
Extensions
Filesystems
Filters
Fonts
Frameworks
Graphics
Image Capture
Input Methods
InternetAccounts
Java
Extensions
JavaConfig.plist
JavaVirtualMachines
Support
1.6.0.jdk
Contents
Classes
Commands
Home
Info.plist
Libraries
MacOS
Resources
version.plist
bin
bund
lib
man
New Folder
Cancel
Open


Eclipse
File
Edit
Navigate
Search
Project
Run
Window
Help
Package Explorer
com.genericworkflownodes.knime
com.genericworkflownodes.knime.feature
com.genericworkflownodes.knime.node_gene
com.genericworkflownodes.knime.tests
de.seqan
de.seqan.feature
de.seqan.macosx.x86_64
Open Project
Close Project
Build All
⌘B
Build Project
Build Working Set
Clean...
Build Automatically
Generate Javadoc...
Properties

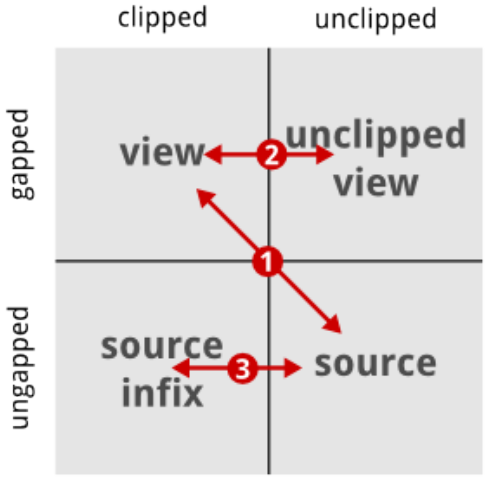

Properties for com.genericworkflownodes.knime.tests
type filter text
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Annotation Processing
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
Plug-in Development
Project References
Run/Debug Settings
Java Compiler
Enable project specific settings
Configure Workspace Settings...
JDK Compliance
Use compliance from execution environment 'JavaSE-1.6' on the 'Java Build Path'
Compiler compliance level:
1.6
Use default compliance settings
Generated .class files compatibility:
1.6
Source compatibility:
1.6
Disallow identifiers called 'assert':
Error
Disallow identifiers called 'enum':
Error
Classfile Generation
Add variable attributes to generated class files (used by the debugger)
Add line number attributes to generated class files (used by the debugger)
Add source file name to generated class file (used by the debugger)
Preserve unused (never read) local variables
Inline finally blocks (larger class files, but improved performance)
Restore Defaults
Apply
Cancel
OK

Computing Positions In Clipped Alignments

This page describes how to compute view and source positions in an unclipped and clipped Align.

Position Computation Overview

There are four coordinate systems related to each gap object. One can consider the positions with and without gaps, both with and without clipping. The following picture and list show the easiest transformations between the coordinate systems.



1. Translate between view (gapped clipped) position and source (ungaped unclipped) position using the functions toSourcePosition and toViewPosition.
2. Translate between clipped and unclipped gapped position by adding/subtracting clippedBeginPosition of the gaps object.
3. Translate between clipped ungapped and unclipped ungapped position by adding/subtracting beginPosition of the gaps object.

All other transformations are most easily done following one of the paths from the picture above.

An Example

The following extensive example shows how to practically translate between the coordinate systems.

```
// Demo program for clipping with Gaps objects.

#include <iostream>

#include <seqan/sequence.h>
#include <seqan/align.h>

using namespace seqan;
```

```
int main()
{
    // Create sequence variable and gaps basd on sequence.
    CharString seq("ABCDEFGHIJ");
    Gaps<CharString> gaps(seq);

    // Insert gaps, the positions are in (clipped) view space.
    insertGaps(gaps, 2, 2);
    insertGap(gaps, 6);
    insertGap(gaps, 10);

    // Print to stdout.
    std::cout << "gaps\t" << gaps << "\n"
              << "seq \t" << seq << "\n\n";

    // Print the begin and end positions in sequence space and the clipped
    // begin and end positions in gap space.  We have no clipping, so no
    // surprises here.
    std::cout << "beginPosition(gaps)        == " << beginPosition(gaps) << "\n"
              << "endPosition(gaps)          == " << endPosition(gaps) << "\n"
              << "clippedBeginPosition(gaps) == " << clippedBeginPosition(gaps) << "\n
↪"
              << "clippedEndPosition(gaps)   == " << clippedEndPosition(gaps) << "\n\n
↪";

    // Now, clip the alignment and again print the gaps, sequence and begin/end
    // positions.  Note that the clipping positions are relative to the unclipped
    // view.
    setClippedBeginPosition(gaps, 3);
    setClippedEndPosition(gaps, 10);

    std::cout << "gaps\t" << gaps << "\n"
              << "seq \t" << infix(seq, beginPosition(gaps), endPosition(gaps)) <<
↪"\n\n";

    std::cout << "beginPosition(gaps)        == " << beginPosition(gaps) << "\n"
              << "endPosition(gaps)          == " << endPosition(gaps) << "\n"
              << "clippedBeginPosition(gaps) == " << clippedBeginPosition(gaps) << "\n
↪"
              << "clippedEndPosition(gaps)   == " << clippedEndPosition(gaps) << "\n\n
↪";

    // We can translate between the (clipped) gapped position (aka view) and
    // the unclipped ungapped positions (aka) source using toSourcePosition()
    // and toViewPosition().  Note that because of projection to the right of
    // gaps, these operations are not symmetric.
    std::cout << "4 view position => " << toSourcePosition(gaps, 4) << " source␣
↪position\n"
              << "2 source position => " << toViewPosition(gaps, 2) << " view␣
↪position\n\n";

    // Translating between clipped gapped and unclipped gapped position can
    // be done by adding/subtracting clippedBeginPosition(gaps).
    std::cout << "3 clipped gapped => " << 3 + clippedBeginPosition(gaps) << "␣
↪unclipped gapped\n"
              << "6 unclipped gapped => " << 5 - clippedBeginPosition(gaps) << "␣
↪clipped gapped\n\n";
```

```
    // Translating between clipped ungapped and unclipped ungapped position can
    // be done by adding/subtracing beginPosition(gaps).  Since there are no
    // gaps, this operation is symmetric.
    std::cout << "3 clipped ungapped => " << 3 + beginPosition(gaps) << " unclipped␣
↪ungapped\n"
              << "5 unclipped ungapped => " << 5 - beginPosition(gaps) << " clipped␣
↪ungapped\n\n";

    // Translating between gapped clipped position and ungapped clipped
    // position and between gapped unclipped and ungapped unclipped positions
    // has to be done using the translations above.
    std::cout << "3 clipped gapped => " << toSourcePosition(gaps, 3) -␣
↪beginPosition(gaps) << " clipped ungapped\n"
              << "4 unclipped ungapped => " << toViewPosition(gaps, 4) +␣
↪clippedBeginPosition(gaps) << " unclipped gapped\n";

    return 0;
}
```

The output looks as follows:

```
gaps        AB--CD-EFG-HIJ
seq         ABCDEFGHIJ

beginPosition(gaps)        == 0
endPosition(gaps)          == 10
clippedBeginPosition(gaps) == 0
clippedEndPosition(gaps)   == 14

gaps        -CD-EFG
seq         CDEFG

beginPosition(gaps)        == 2
endPosition(gaps)          == 7
clippedBeginPosition(gaps) == 3
clippedEndPosition(gaps)   == 10

4 view position => 4 source position
2 source position => 1 view position

3 clipped gapped => 6 unclipped gapped
6 unclipped gapped => 2 clipped gapped

3 clipped ungapped => 5 unclipped ungapped
5 unclipped ungapped => 3 clipped ungapped

3 clipped gapped => 2 clipped ungapped
4 unclipped ungapped => 7 unclipped gapped
```

ToC

Contents

Filtering Similar Sequences

Using Swift

In the next example we are going to use the **Swift** filter to efficiently find pairs of similar reads. The Swift algorithm searches for so-called epsilon matches (local alignments) of two sequences with an error rate below a certain epsilon threshold.

The Swift implementation in SeqAn provides a find interface and requires the Finder and Pattern to be specialized with `Swift<..>`. Millions of sequences can be searched simultaneously with one Swift Pattern in a Swift Finder of a single haystack sequence. The error rate of a local alignment is the number of errors divided by the length of the needle sequence part of the match. There are currently two version of the Swift algorithm implemented in SeqAn: `SwiftSemiGlobal` and `SwiftLocal`. Both can be used to search epsilon-matches of a certain minimum length.

Hint: `SwiftSemiGlobal` should only be used for short needles (sequenced reads) as it always returns potential epsilon matches spanning a whole needle sequence. `SwiftLocal` should be preferred for large needles as it returns needle sequences potentially having an intersection with an epsilon match.

The following program searches for semi-global alignments between pairs of reads with a maximal error rate of 10%.

```
#include <seqan/stream.h>
#include <seqan/index.h>
#include <seqan/store.h>
#include <iostream>

using namespace seqan;
```

First we loads reads from a file into a FragmentStore with loadReads.

```
int main(int argc, char const * argv[])
{
    FragmentStore<> fragStore;
    if (argc < 2 || !loadReads(fragStore, argv[1]))
    {
        std::cerr << "ERROR: Coud not load reads." << std::endl;
        return 0;
    }
```

Swift uses a q-gram index of the needle sequences. Thus, we have to specialize the Swift Semi Global Pattern with a IndexQGram index of the needle StringSet in the first template argument, create the index over the readSeqStore and pass the index to the Pattern constructor. Swift Semi Global Finder and Swift Semi Global Pattern classes have to be specialized with `SwiftSemiGlobal` in the second template argument.

Note: Note, to use the local swift filter you simply switch the specialization tag to `SwiftLocal`: Swift Local Finder and Swift Local Pattern.

The main loop iterates over all potential matches which can be further processed, e.g. by a semi-global or overlap aligner.

```
    typedef FragmentStore<>::TReadSeqStore TReadSeqStore;
    typedef GetValue<TReadSeqStore>::Type TReadSeq;
    typedef Index<TReadSeqStore, IndexQGram<Shape<Dna, UngappedShape<11> >,␣
↪OpenAddressing> > TIndex;
    typedef Pattern<TIndex, Swift<SwiftSemiGlobal> > TPattern;
```

```
    typedef Finder<TReadSeq, Swift<SwiftSemiGlobal> > TFinder;

    TIndex index(fragStore.readSeqStore);
    TPattern pattern(index);
    for (unsigned i = 0; i < length(fragStore.readSeqStore); ++i)
    {
        if ((i % 1000) == 0)
            std::cout << "." << std::flush;
        TFinder finder(fragStore.readSeqStore[i]);
        while (find(finder, pattern, 0.1))
        {
            if (i == position(pattern).i1)
                continue;
            // do further alignment here
/*                      std::cout << "Found possible overlap of " << std::endl;
            std::cout << "\t" << infix(finder) << std::endl;
            std::cout << "\t" << seqs[position(pattern).i1] << std::endl;
*/      }
    }

    return 0;
}
```

ToC

Contents

Metafunctions

Type Metafunctions

For example, the metafunction Iterator is a type metafunction, i.e. it is used to determine a type. Type metafunctions have the form:

`typename TypeMetaFunc<T1, T2, ..., TN>::Type`

`TypeMetaFunc` The name of the metafunction

`T1, T2, ..., TN` Arguments (types or constants)

`Type` The resulting type

The keyword `typename` must be stated if one of the arguments `T1, T2, ..., TN` is or uses a template parameter. For example the following piece of code uses the metafunction `Iterator` to determine an iterator type for a string class:

```
    String<char> str = "I am a string";
    Iterator<String<char> >::Type it = begin(str);
    while (! atEnd(it, str))
    {
```

```
        std::cout << *it;
        ++it;
    }
    std::cout << std::endl;
```

```
I am a string
```

Value Metafunctions

Metafunctions can also be used to determine constant values at compile time. The general form of value metafunctions is:

`VALUE_META_FUNC<T1, T2, ..., TN>::VALUE`

`VALUE_META_FUNC` The name of the metafunction

`T1, T2, ..., TN` Arguments (types or constants)

`VALUE` The resulting constant value

For example the following function prints the length of a fixed sized string using the value metafunction LENGTH:

```
template <typename T>
void printLenOfFixedSizeString(T const &)
{
    std::cout << LENGTH<T>::VALUE << std::endl;
}

int main()
{
    String<char, Array<100> > my_str;
    printLenOfFixedSizeString(my_str);
    return 0;
}
```

```
100
```

Important: Different uses of "*Value*":

Please note that Value (`Value<TSomeType>::Type`) is a **Type Metafunction**, because it returns a Type (e.g. of values in a container) and not a value.

Assignment 1

Objective Write a generic function `checkContainerForDna(T & container)` that prints out a message if the value inside the container is of the type Dna. The type `T` of the container should be specified as a template argument. Test you function with some examples.

Hint

- Use the **Type Metafunction** Value to access the (alphabet-)type of the elements in the container.
- Use the **Value Metafunction** IsSameType to check for type equality.

Solution Your program should look something like this:

```
#include <iostream>
#include <seqan/basic.h>
#include <seqan/stream.h>

using namespace seqan;

template <typename T>
void checkContainerForDna(T & container)
{
    // Type Metafunction Value<>
    typedef typename Value<T>::Type TAlphType;

    // Value Metafunction IsSameType<> evaluated at compile time
    if (IsSameType<TAlphType, Dna>::VALUE)
        std::cout << "I have found a container with Dna!" <<  std::endl;
    else
        std::cout << "No Dna anywhere." <<  std::endl;
}

int main()
{
    typedef String<Dna> TDnaString;
    TDnaString dna = "AAAATTTT";

    typedef String<int> TIntString;

    TIntString numbers;
    appendValue(numbers, 1);
    appendValue(numbers, 3);

    checkContainerForDna(dna);
    checkContainerForDna(numbers);

    return 0;
}
```

Note: Because the Value Metafunction `IsSameType<>` is evaluated at compile time, the part of the if-statement code that does not apply won't even appear in the compiled code. This can be an improvement to the runtime of your code.

The output is the following:

```
I have found a container with Dna!
No Dna anywhere.
```

ToC

Contents

Working With Custom Score Matrices

This How To describes how to create new scoring matrices for Amino Acids and DNA alphabets and how to load score matrices from files.

Creating A New Built-In Score Matrix

The following program demonstrates how to implement a new built-in score matrix.

```
#include <iostream>

#include <seqan/basic.h>
#include <seqan/stream.h>   // For printing strings.
#include <seqan/score.h>    // The module score.

using namespace seqan;
```

We need to perform the necessary definitions for the matrix. This consists of two steps:

1. Defining a tag struct.
2. Specializing the class `ScoringMatrixData_` with your tag.

Note how we use enum values to compute the matrix size which itself is retrieved from the ValueSize metafunction.

```
// Extend SeqAn by a user-define scoring matrix.
namespace seqan {

// We have to create a new specialization of the ScoringMatrix_ class
// for the DNA alphabet.  For this, we first create a new tag.
struct UserDefinedMatrix {};

// Then, we specialize the class ScoringMatrix_ for the Dna5 alphabet.
template <>
struct ScoringMatrixData_<int, Dna5, UserDefinedMatrix>
{
    enum
    {
        VALUE_SIZE = ValueSize<Dna5>::VALUE,
        TAB_SIZE = VALUE_SIZE * VALUE_SIZE
    };

    static inline int const * getData()
    {
        // The user defined data table.  In this case, we use the data from BLOSUM-30.
        static int const _data[TAB_SIZE] =
        {
            1, 0, 0, 0, 0,
            0, 1, 0, 0, 0,
            0, 0, 1, 0, 0,
            0, 0, 0, 1, 0,
            0, 0, 0, 0, 0
        };
        return _data;
    }

};
}  // namespace seqan
```

Now we define a function `showScoringMatrix` for displaying a matrix.

```
// Print a scoring scheme matrix to stdout.
template <typename TScoreValue, typename TSequenceValue, typename TSpec>
void showScoringMatrix(Score<TScoreValue, ScoreMatrix<TSequenceValue, TSpec> > const &
↪ scoringScheme)
{
    // Print top row.
    for (unsigned i = 0; i < ValueSize<TSequenceValue>::VALUE; ++i)
        std::cout << "\t" << TSequenceValue(i);
    std::cout << std::endl;
    // Print each row.
    for (unsigned i = 0; i < ValueSize<TSequenceValue>::VALUE; ++i)
    {
        std::cout << TSequenceValue(i);
        for (unsigned j = 0; j < ValueSize<TSequenceValue>::VALUE; ++j)
        {
            std::cout << "\t" << score(scoringScheme, TSequenceValue(i),␣
↪TSequenceValue(j));
        }
        std::cout << std::endl;
    }
}
```

Finally, the `main` function demostrates some of the things you can do with scores:

- Construct an empty score matrix object (2.)
- Fill the score matrix in a loop (3.1)
- Fill the matrix with the user-defined matrix values (3.2)
- Directly create a score matrix with the user-defined matrix values (4)

```
int main()
{
    // 1. Define type and constants.
    //
    // Define types for the score value and the scoring scheme.
    typedef int TValue;
    typedef Score<TValue, ScoreMatrix<Dna5, Default> > TScoringScheme;
    // Define our gap scores in some constants.
    int const gapOpenScore = -1;
    int const gapExtendScore = -1;

    // 2. Construct scoring scheme with default/empty matrix.
    //
    // Construct new scoring scheme, alternatively only give one score
    // that is used for both opening and extension.
    TScoringScheme scoringScheme(gapExtendScore, gapOpenScore);

    // 3. Fill the now-existing ScoreMatrix
    //
    // The scoring scheme now already has a matrix of the size
    // ValueSize<Dna5>::VALUE x ValueSize<Dna5>::VALUE which
    // we can now fill.

    // 3.1 We fill the scoring scheme with the product of the coordinates.
    std::cout << std::endl << "Coordinate Products" << std::endl;
    for (unsigned i = 0; i < ValueSize<Dna5>::VALUE; ++i)
```

```
    {
        for (unsigned j = 0; j < ValueSize<Dna5>::VALUE; ++j)
        {
            setScore(scoringScheme, Dna5(i), Dna5(j), i * j);
        }
    }
    showScoringMatrix(scoringScheme);

    // 3.2 Now, we fill it with the user defined matrix above.
    std::cout << "User defined matrix (also Dna5 scoring matrix)..." << std::endl;
    setDefaultScoreMatrix(scoringScheme, UserDefinedMatrix());
    showScoringMatrix(scoringScheme);

    // 4. Show our user-defined Dna5 scoring matrix.
    std::cout << "User DNA scoring scheme..." << std::endl;
    Score<TValue, ScoreMatrix<Dna5, UserDefinedMatrix> > userScoringSchemeDna;
    showScoringMatrix(userScoringSchemeDna);

    return 0;
}
```

Here is the output of the program:

```
Coordinate Products
        A        C        G        T        N
A        0        0        0        0        0
C        0        1        2        3        4
G        0        2        4        6        8
T        0        3        6        9        12
N        0        4        8        12        16
User defined matrix (also Dna5 scoring matrix)...
        A        C        G        T        N
A        1        0        0        0        0
C        0        1        0        0        0
G        0        0        1        0        0
T        0        0        0        1        0
N        0        0        0        0        0
User DNA scoring scheme...
        A        C        G        T        N
A        1        0        0        0        0
C        0        1        0        0        0
G        0        0        1        0        0
T        0        0        0        1        0
N        0        0        0        0        0
```

Loading Score Matrices From File

This small demo program shows how to load a score matrix from a file. Examples for a score file are `demos/howto/scores/dna_example.txt` for DNA alphabets and `tests/sPAM250` for amino acids.

Include the necessary headers.

```
#include <iostream>

#include <seqan/basic.h>
#include <seqan/stream.h>   // For printing strings.
```

```
#include <seqan/score.h>  // The module score.

using namespace seqan;
```

We define a function that can show a scoring matrix.

```
// Print a scoring scheme matrix to stdout.
template <typename TScoreValue, typename TSequenceValue, typename TSpec>
void showScoringMatrix(Score<TScoreValue, ScoreMatrix<TSequenceValue, TSpec> > const &
↪ scoringScheme)
{
    // Print top row.
    for (unsigned i = 0; i < ValueSize<TSequenceValue>::VALUE; ++i)
        std::cout << "\t" << TSequenceValue(i);
    std::cout << std::endl;
    // Print each row.
    for (unsigned i = 0; i < ValueSize<TSequenceValue>::VALUE; ++i)
    {
        std::cout << TSequenceValue(i);
        for (unsigned j = 0; j < ValueSize<TSequenceValue>::VALUE; ++j)
        {
            std::cout << "\t" << score(scoringScheme, TSequenceValue(i),␣
↪TSequenceValue(j));
        }
        std::cout << std::endl;
    }
}
```

Finally, the main program loads the scoring matrix and then shows it.

```
int main(int argc, char ** argv)
{
    typedef int TScoreValue;

    Score<TScoreValue, ScoreMatrix<Dna5, Default> > scoreMatrix;
    loadScoreMatrix(scoreMatrix, toCString(getAbsolutePath("demos/howto/scores/dna_
↪example.txt")));
    showScoringMatrix(scoreMatrix);

    return 0;
}
```

Here's the program output.

```
        A        C        G        T        N
A        1       -1       -1       -1       -1
C       -1        1       -1       -1       -1
G       -1       -1        1       -1       -1
T       -1       -1       -1        1       -1
N       -1       -1       -1       -1        1
```

Use Cases

After reading the Tutorial we recommend to read some of the following use cases:

ToC

Contents

Simple Overlapper

ToC

Contents

Simple RNA-Seq

Learning Objective You will learn how to write a simple gene quantification tool based on RNA-Seq data.

Difficulty Hard

Duration 2h

Prerequisites *Genome Annotations*, *Fragment Store*, experience with OpenMP (optional)

RNA-Seq refers to high-throughput sequencing of cDNA in order to get information about the RNA molecules available in a sample. Knowing the sequence and abundance of mRNA allows to determine the (differential) expression of genes, to detect alternative splicing variants, or to annotate yet unknown genes.

In the following tutorial you will develop a simple gene quantification tool. It will load a file containing gene annotations and a file with RNA-Seq read alignments, compute abundances, and output RPKM values for each expressed gene.

Albeit its simplicity, this example can be seen as a starting point for more complex applications, e.g. to extend the tool from quantification of genes to the quantification of (alternatively spliced) isoforms, or to de-novo detect yet unannotated isoforms/genes.

You will learn how to use the FragmentStore to access gene annotations and alignments and how to use the IntervalTree to efficiently determine which genes overlap a read alignment.

Introduction to the used Data Structures

This section introduces the FragmentStore and the IntervalTree, which are the fundamental data structures used in this tutorial to represent annotations and read alignments and to efficiently find overlaps between them. You may skip one or both subsections if you are already familiar with one or both data structures.

Fragment Store

The FragmentStore is a data structure specifically designed for read mapping, genome assembly or gene annotation. These tasks typically require lots of data structures that are related to each other such as

- reads, mate-pairs, reference genome
- pairwise alignments, and
- genome annotation.

The Fragment Store subsumes all these data structures in an easy to use interface. It represents a multiple alignment of millions of reads or mate-pairs against a reference genome consisting of multiple contigs. Additionally, regions of the reference genome can be annotated with features like ‘gene’, ‘mRNA’, ‘exon’, ‘intron’ (see PredefinedAnnotationTypes) or custom features. The Fragment Store supports I/O functionality to read/write a read alignment in SAM or AMOS format and to read/write annotations in GFF or GTF format.

The Fragment Store can be compared with a database where each table (called “store”) is implemented as a String member of the FragmentStore class. The rows of each table (implemented as structs) are referred by their ids which are their positions in the string and not stored explicitly. The only exception is the alignedReadStore whose elements of type AlignedReadStoreElement contain an id-member as they may be rearranged in arbitrary order, e.g. by increasing genomic positions or by readId. Many stores have an associated name store to store element names. Each name store is a StringSet that stores the element name at the position of its id. All stores are present in the Fragment Store and empty if unused. The concrete types, e.g. the position types or read/contig alphabet, can be easily changed by defining a custom config struct which is a template parameter of the Fragment Store class.

Annotation Tree

Annotations are represented as a tree that at least contains a root node where all annotations are children or grandchildren of. A typical annotation tree looks as follows:

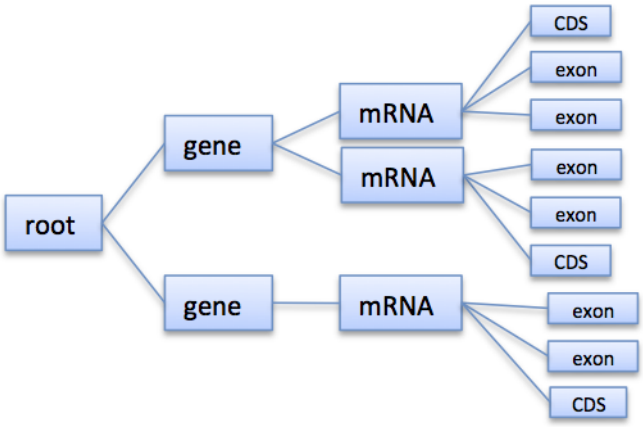


Fig. 4.9: Annotation tree example

In the Fragment Store the tree is represented by annotationStore, annotationTypeStore, annotationKeyStore, and others. Instead of accessing these tables directly, the AnnotationTree Iterator provides a high-level interface to traverse and access the annotation tree.

Interval Tree

The IntervalTree is a data structure that stores one-dimensional intervals in a balanced tree and efficiently answers range queries. A range query is an operation that returns all tree intervals that overlap a given query point or interval.

The interval tree implementation provided in SeqAn is based on a Tree which is balanced if all intervals are given at construction time. Interval tree nodes are objects of the IntervalAndCargo class and consist of 2 interval boundaries and additional user-defined information, called cargo. To construct the tree on a set of given interval nodes use the function createIntervalTree. The functions addInterval and removeInterval should only be used if the interval tree needs to be changed dynamically (as they not yet balance the tree).

Import Alignments and Gene Annotations from File

At first, our application should create an empty `FragmentStore` object into which we import a gene annotation file and a file with RNA-Seq alignments. An empty `FragmentStore` can simply be created with:

```
FragmentStore<> store;
```

Files can be read from disk with the function read that expects an open stream (e.g. a STL ifstream), a `FragmentStore` object, and a File Format tag. The contents of different files can be loaded with subsequent calls of `read`. As we want the user to specify the files via command line, our application will parse them using the ArgumentParser and store them in an option object.

In your first assignment you need to complete a given code template and implement a function that loads a SAM file and a GTF file into the `FragmentStore`.

Assignment 1

Type Application

Objective Use the code template below (click **more...**) and implement the function `loadFiles` to load the annotation and alignment files. Use the file paths given in the options object and report an error if the files could not be opened.

```
#include <iostream>
#include <seqan/store.h>
#include <seqan/misc/interval_tree.h>
#include <seqan/parallel.h>

using namespace seqan;


// define used types
typedef FragmentStore<> TStore;
//
// 2. Load annotations and alignments from files
//
bool loadFiles(TStore & store, std::string const & annotationFileName, ␣
↪std::string const & alignmentFileName)
{
    // INSERT YOUR CODE HERE ...
    //

    return true;
}

//![main]
int main(int argc, char const * argv[])
{
    TStore store;
    std::string annotationFileName = getAbsolutePath("demos/tutorial/simple_rna_
↪seq/example.gtf");
    std::string alignmentFileName = getAbsolutePath("demos/tutorial/simple_rna_
↪seq/example.sam");

    if (!loadFiles(store, annotationFileName, annotationFileName))
        return 1;

    return 0;
}
//![main]
```

Hint

- Open STL std::fstream objects and use the function read with a SAM or GTF tag.
- ifstream::open requires the file path to be given as a C-style string (`const char *`).
- Use string::c_str to convert the option strings into C-style strings.
- The function read expects a stream, a FragmentStore and a tag, i.e. `Sam()` or `Gtf()`.

Solution

```
//
// 1. Load annotations and alignments from files
```

```
//
bool loadFiles(TStore & store, std::string const & annotationFileName, ␣
↪std::string const & alignmentFileName)
{
    BamFileIn alignmentFile;
    if (!open(alignmentFile, alignmentFileName.c_str()))
    {
        std::cerr << "Couldn't open alignment file " << alignmentFileName <<␣
↪std::endl;
        return false;
    }
    std::cout << "Loading read alignments ..... " << std::flush;
    readRecords(store, alignmentFile);
    std::cout << "[" << length(store.alignedReadStore) << "]" << std::endl;

    // load annotations
    GffFileIn annotationFile;
    if (!open(annotationFile, toCString(annotationFileName)))
    {
        std::cerr << "Couldn't open annotation file" << annotationFileName <<␣
↪std::endl;
        return false;
    }
    std::cout << "Loading genome annotation ... " << std::flush;
    readRecords(store, annotationFile);
    std::cout << "[" << length(store.annotationStore) << "]" << std::endl;

    return true;
}
```

Extract Gene Intervals

Now that the Fragment Store contains the whole annotation tree, we want to traverse the genes and extract the genomic ranges they span. In the annotation tree, genes are (the only) children of the root node. To efficiently retrieve the genes that overlap read alignments later, we want to use interval trees, one for each contig. To construct an interval tree, we first need to collect IntervalAndCargo objects in a string and pass them to createIntervalTree. See the interval tree demo in `demos/interval_tree.cpp` for more details. As cargo we use the gene's annotation id to later retrieve all gene specific information. The strings of `IntervalAndCargo` objects should be grouped by `contigId` and stored in an (outer) string of strings. For the sake of simplicity we don't differ between genes on the forward or reverse strand and instead always consider the corresponding intervals on the forward strand.

To define this string of strings of `IntervalAndCargo` objects, we first need to determine the types used to represent an annotation. All annotations are stored in the annotationStore which is a Fragment Store member and whose type is TAnnotationStore. The value type of the annotation store is the class AnnotationStoreElement. Its member typedefs TPos and TId define the types it uses to represent a genomic position or the annotation or contig id:

```
    typedef FragmentStore<> TStore;
    typedef Value<TStore::TAnnotationStore>::Type TAnnotation;
    typedef TAnnotation::TId TId;
    typedef TAnnotation::TId TPos;
    typedef IntervalAndCargo<TPos, TId> TInterval;
```

The string of strings of intervals can now be defined as:

```
    String<String<TInterval> > intervals;
```

In your second assignment you should use an AnnotationTree Iterator to traverse all genes in the annotation tree. For each gene, determine its genomic range (projected to the forward strand) and add a new `TInterval` object to the `intervals[contigId]` string, where `contigId` is the id of the contig containing that gene.

Assignment 2

Type Application

Objective Use the code template below (click **more..**). Implement the function `extractGeneIntervals` that should extract genes from the annotation tree (see AnnotationTree Iterator) and create strings of IntervalAndCargo objects - one for each config - that contains the interval on the forward contig strand and the gene's annotation id.

Extend the definitions:

```
// define used types
typedef FragmentStore<>                             TStore;
typedef Value<TStore::TAnnotationStore>::Type       TAnnotation;
typedef TAnnotation::TId                            TId;
typedef TAnnotation::TPos                           TPos;
typedef IntervalAndCargo<TPos, TId>                 TInterval;
```

Add a function:

```
//
// 3. Extract intervals from gene annotations (grouped by contigId)
//
void extractGeneIntervals(String<String<TInterval> > & intervals, TStore const &␣
↪store)
{
    // INSERT YOUR CODE HERE ...
    //
}
```

Extend the `main` function:

```
    TStore store;
    String<String<TInterval> > intervals;
```

and

```
    extractGeneIntervals(intervals, store);
```

Hint You can assume that all genes are children of the root node, i.e. create an AnnotationTree Iterator, go down to the first gene and go right to visit all other genes. Use getAnnotation to access the gene annotation and value to get the annotation id.

Make sure that you append IntervalAndCargo objects, where `i1` < `i2` holds, as opposed to annotations where `beginPos` > `endPos` is possible. Remember to ensure that `intervals` is of appropriate size, e.g. with

```
    resize(intervals, length(store.contigStore));
```

Use appendValue to add a new `TInverval` object to the inner string, see IntervalAndCargo constructor for the constructor.

Solution

```
//
// 2. Extract intervals from gene annotations (grouped by contigId)
//
void extractGeneIntervals(String<String<TInterval> > & intervals, TStore const &␣
↪store)
{
    // extract intervals from gene annotations (grouped by contigId)
    resize(intervals, length(store.contigStore));

    Iterator<TStore const, AnnotationTree<> >::Type it = begin(store,␣
↪AnnotationTree<>());

    if (!goDown(it))
        return;

    do
    {
        SEQAN_ASSERT_EQ(getType(it), "gene");
        TPos beginPos = getAnnotation(it).beginPos;
        TPos endPos = getAnnotation(it).endPos;
        TId contigId = getAnnotation(it).contigId;

        if (beginPos > endPos)
            std::swap(beginPos, endPos);

        // insert forward-strand interval of the gene and its annotation id
        appendValue(intervals[contigId], TInterval(beginPos, endPos, value(it)));
    }
    while (goRight(it));
}
```

Construct Interval Trees

With the strings of gene intervals - one for each contig - we now can construct interval trees. Therefore, we specialize an IntervalTree with the same position and cargo types as used for the IntervalAndCargo objects. As we need an interval tree for each contig, we instantiate a string of interval trees:

```
    typedef IntervalTree<TPos, TId> TIntervalTree;
    String<TIntervalTree> intervalTrees;
```

Your third assignment is to implement a function that constructs the interval trees for all contigs given the string of interval strings.

Assignment 3

Type Application

Objective Use the code template below (click **more...**). Implement the function `constructIntervalTrees` that uses the interval strings to construct for each contig an interval tree. **Optional:** Use OpenMP to parallelize the construction over the contigs, see SEQAN_OMP_PRAGMA.

Extend the definitions:

```
// define used types
typedef FragmentStore<>                         TStore;
```

```
typedef Value<TStore::TAnnotationStore>::Type   TAnnotation;
typedef TAnnotation::TId                        TId;
typedef TAnnotation::TPos                       TPos;
typedef IntervalAndCargo<TPos, TId>             TInterval;
typedef IntervalTree<TPos, TId>                 TIntervalTree;
```

Add a function:

```
//
// 3. Construct interval trees
//
void constructIntervalTrees(String<TIntervalTree> & intervalTrees,
                            String<String<TInterval> > & intervals)
{
    // INSERT YOUR CODE HERE ...
    //
}
```

Extend the `main` function:

```
    String<String<TInterval> > intervals;
    String<TIntervalTree> intervalTrees;
```

and

```
    extractGeneIntervals(intervals, store);
    constructIntervalTrees(intervalTrees, intervals);
```

Hint First, resize the string of interval trees accordingly:

```
    resize(intervals, length(store.contigStore));
```

Hint Use the function createIntervalTree.

Optional: Construct the trees in parallel over all contigs with an OpenMP parallel for-loop, see here for more information about OpenMP.

Solution

```
//
// 3. Construct interval trees
//
void constructIntervalTrees(String<TIntervalTree> & intervalTrees,
                            String<String<TInterval> > & intervals)
{
    int numContigs = length(intervals);
    resize(intervalTrees, numContigs);

    SEQAN_OMP_PRAGMA(parallel for)
    for (int i = 0; i < numContigs; ++i)
        createIntervalTree(intervalTrees[i], intervals[i]);
}
```

Compute Gene Coverage

To determine gene expression levels, we first need to compute the read coverage, i.e. the total number of reads overlapping a gene. Therefore we use a string of counters addressed by the annotation id.

```
    String<unsigned> readsPerGene;
```

For each read alignment we want to determine the overlapping genes by conducting a range query via findIntervals and then increment their counters by 1. To address the counter of a gene, we use its annotation id stored as cargo in the interval tree.

Read alignments are stored in the alignedReadStore, a string of AlignedReadStoreElements objects. Their actual type can simply be determined as follows:

```
    typedef Value<TStore::TAlignedReadStore>::Type TAlignedRead;
```

Given the contigId, beginPos, and endPos we will retrieve the annotation ids of overlapping genes from the corresponding interval tree.

Your fourth assignment is to implement the count function that performs all the above described steps. Optionally, use OpenMP to parallelize the counting.

Assignment 4

Type Application

Objective Use the code template below (click **more...**). Implement the function `countReadsPerGene` that counts for each gene the number of overlapping reads. Therefore determine for each AlignedReadStoreElement begin and end positions (on forward strand) of the alignment and increment the `readsPerGene` counter for each overlapping gene.

Optional: Use OpenMP to parallelize the function, see SEQAN_OMP_PRAGMA.

Extend the definitions:

```
// define used types
typedef FragmentStore<>                          TStore;
typedef Value<TStore::TAnnotationStore>::Type    TAnnotation;
typedef TAnnotation::TId                         TId;
typedef TAnnotation::TPos                        TPos;
typedef IntervalAndCargo<TPos, TId>              TInterval;
typedef IntervalTree<TPos, TId>                  TIntervalTree;
typedef Value<TStore::TAlignedReadStore>::Type   TAlignedRead;
```

Add a function:

```
//
// 4. Count reads per gene
//
void countReadsPerGene(String<unsigned> & readsPerGene, String<TIntervalTree>␣
↪const & intervalTrees, TStore const & store)
{
    // INSERT YOUR CODE HERE ...
    //
}
```

Extend the `main` function:

```
    String<TIntervalTree> intervalTrees;
    String<unsigned> readsPerGene;
```

and

```
    extractGeneIntervals(intervals, store);
    constructIntervalTrees(intervalTrees, intervals);
    countReadsPerGene(readsPerGene, intervalTrees, store);
```

Hint

```
    resize(readsPerGene, length(store.annotationStore), 0);
```

Make sure that you search with findIntervals where `query_begin < query_end` holds, as opposed to read alignments where `beginPos > endPos` is possible.

Hint The result of a range query is a string of annotation ids given to findIntervals by-reference:

```
    String<TId> result;
```

Reuse the result string for multiple queries (of the same thread, use `private(result)` for OpenMP).

Solution

```
//
// 4. Count reads per gene
//
void countReadsPerGene(String<unsigned> & readsPerGene, String<TIntervalTree>␣
↪const & intervalTrees, TStore const & store)
{
    resize(readsPerGene, length(store.annotationStore), 0);
    String<TId> result;
    int numAlignments = length(store.alignedReadStore);

    // iterate aligned reads and get search their begin and end positions
    SEQAN_OMP_PRAGMA(parallel for private (result))
    for (int i = 0; i < numAlignments; ++i)
    {
        TAlignedRead const & ar = store.alignedReadStore[i];
        TPos queryBegin = _min(ar.beginPos, ar.endPos);
        TPos queryEnd = _max(ar.beginPos, ar.endPos);

        // search read-overlapping genes
        findIntervals(result, intervalTrees[ar.contigId], queryBegin, queryEnd);

        // increase read counter for each overlapping annotation given the id in␣
↪the interval tree
        for (unsigned j = 0; j < length(result); ++j)
        {
            SEQAN_OMP_PRAGMA(atomic)
            readsPerGene[result[j]] += 1;
        }
    }
}
```

Output RPKM Values

In the final step, we want to output the gene expression levels in a normalized measure. We therefore use **RPKM** values, i.e. the number of **r**eads **p**er **k**ilobase of exon model per **m**illion mapped reads (1). One advantage of RPKM values is their independence of the sequencing throughput (normalized by total mapped reads), and that they allow to compare the expression of short with long transcripts (normalized by exon length).

The exon length of an mRNA is the sum of lengths of all its exons. As a gene may have multiple mRNA, we will simply use the maximum of all their exon lengths.

Your final assignment is to output the RPKM value for genes with a read counter > 0. To compute the exon length of the gene (maximal exon length of all mRNA) use an AnnotationTree Iterator and iterate over all mRNA (children of the gene) and all exons (children of mRNA). For the number of total mapped reads simply use the number of alignments in the alignedReadStore. Output the gene names and their RPKM values separated by tabs as follows:

```
Loading read alignments ..... [22]
Loading genome annotation ... [12]
#gene name        RPKM value
```

Download and decompress the attached mouse annotation (Mus_musculus.NCBIM37.61.gtf.zip and the alignment file of RNA-Seq reads aligned to chromosome Y (sim40mio_onlyY.sam.zip). Test your program and compare your output with the output above.

Assignment 5

Type Application

Objective Use the code template below (click **more...**). Implement the function `outputGeneCoverage` that outputs for each expressed gene the gene name and the expression level as RPKM as tab-separated values.

Add a function:

```
//
// 5. Output RPKM values
//
void outputGeneCoverage(String<unsigned> const & readsPerGene, TStore const &␣
↪store)
{
    // INSERT YOUR CODE HERE ...
    //
}
```

Extend the `main` function:

```
    extractGeneIntervals(intervals, store);
    constructIntervalTrees(intervalTrees, intervals);
    countReadsPerGene(readsPerGene, intervalTrees, store);
    outputGeneCoverage(readsPerGene, store);
```

Hint To compute the maximal exon length use three nested loops: (1) enumerate all genes, (2) enumerate all mRNA of the gene, and (3) enumerate all exons of the mRNA and sum up their lengths.

Hint Remember that exons are not the only children of mRNA.

Solution

```
//
// 5. Output RPKM values
//
void outputGeneCoverage(String<unsigned> const & readsPerGene, TStore const &␣
↪store)
{
    // output abundances for covered genes
    Iterator<TStore const, AnnotationTree<> >::Type transIt = begin(store,␣
↪AnnotationTree<>());
```

```
    Iterator<TStore const, AnnotationTree<> >::Type exonIt;
    double millionMappedReads = length(store.alignedReadStore) / 1000000.0;

    std::cout << "#gene name\tRPKM value" << std::endl;
    for (unsigned j = 0; j < length(readsPerGene); ++j)
    {
        if (readsPerGene[j] == 0)
            continue;

        unsigned mRNALengthMax = 0;
        goTo(transIt, j);

        // determine maximal mRNA length (which we use as gene length)
        SEQAN_ASSERT_NOT(isLeaf(transIt));
        goDown(transIt);

        do
        {
            exonIt = nodeDown(transIt);
            unsigned mRNALength = 0;

            // determine mRNA length, sum up the lengths of its exons
            do
            {
                if (getAnnotation(exonIt).typeId == store.ANNO_EXON)
                    mRNALength += abs((int)getAnnotation(exonIt).beginPos -␣
↪(int)getAnnotation(exonIt).endPos);
            }
            while (goRight(exonIt));

            if (mRNALengthMax < mRNALength)
                mRNALengthMax = mRNALength;
        }
        while (goRight(transIt));

        // RPKM is number of reads mapped to a gene divided by its gene length in␣
↪kbps
        // and divided by millions of total mapped reads
        std::cout << store.annotationNameStore[j] << '\t';
        std::cout << readsPerGene[j] / (mRNALengthMax / 1000.0) /␣
↪millionMappedReads << std::endl;
    }
}
```

Next Steps

- See *[MWM+08]* for further reading.
- Read the *SAM and BAM I/O* Tutorial and change your program to stream a SAM file instead of loading it as a whole.
- Change the program such that it attaches the RPKM value as a key-value pair (see assignValueByKey) to the annotation of each gene and output a GFF file.
- Continue with the *Tutorials* rest of the tutorials]].

For experienced SeqAn developers we gathered some recipes and use cases that might become handy in some situa-

tions. Feel free to contact us if you have a great recipe or a nice use case that uses the SeqAn library.

Workflows

ToC

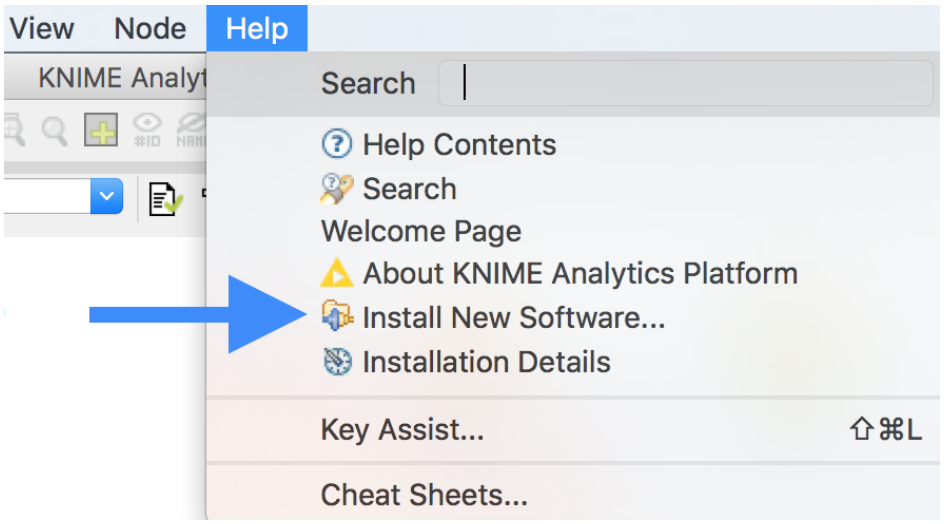
Contents

Creating Workflows with SeqAn Nodes in KNIME

KNIME is a well-established data analysis framework which supports the generation of workflows for data analysis. In this tutorial, we describe how to use SeqAn applications in KNIME.

Install SeqAn in KNIME

The Installation of the SeqAn NGS Toolbox in KNIME is very easy. Download the latest KNIME release from the KNIME website download page. You might be asked for registration but that is optional. In the KNIME window, click on the menu `Help > Install new Software`.

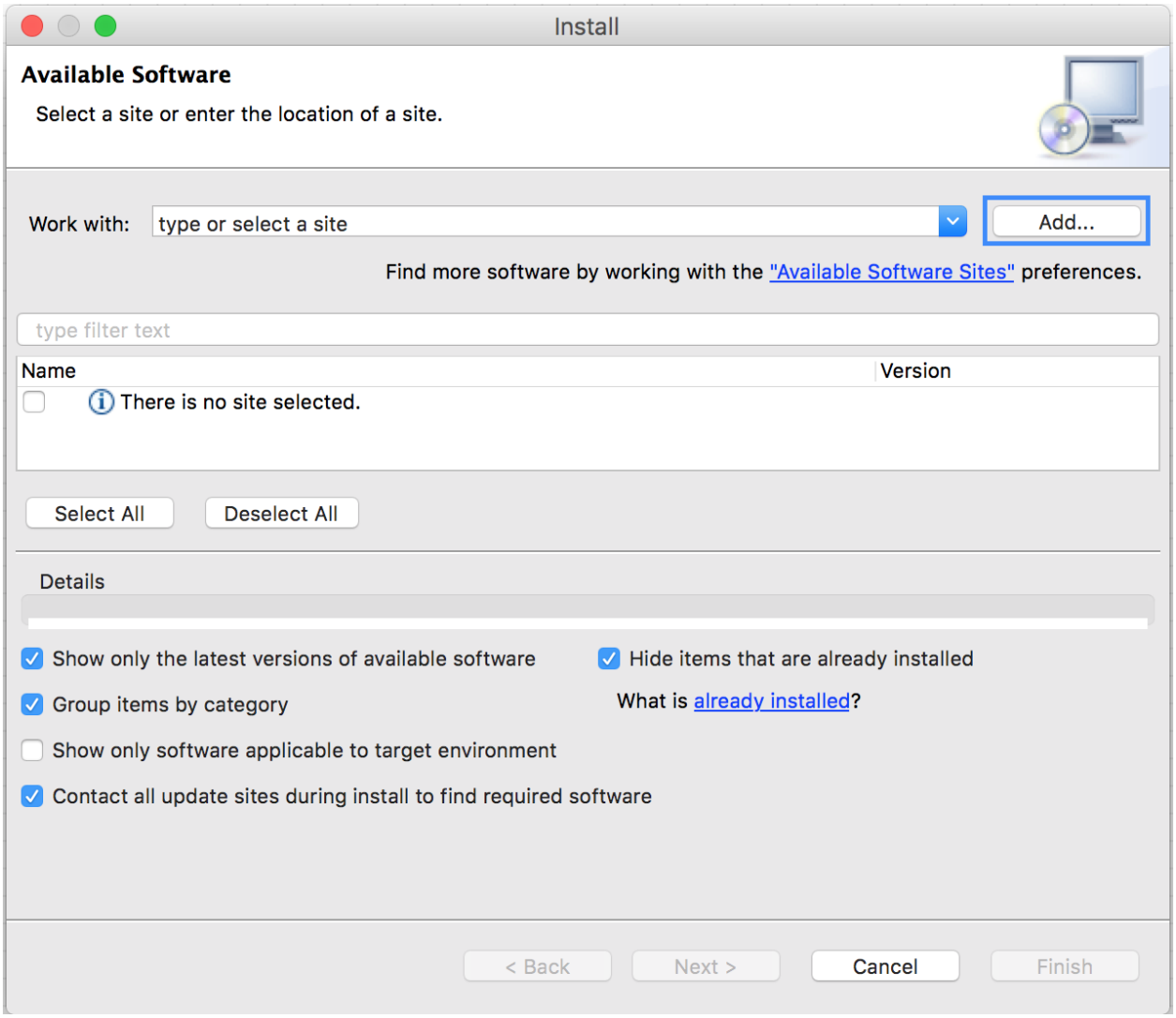


In the opening dialog choose `Add...`.

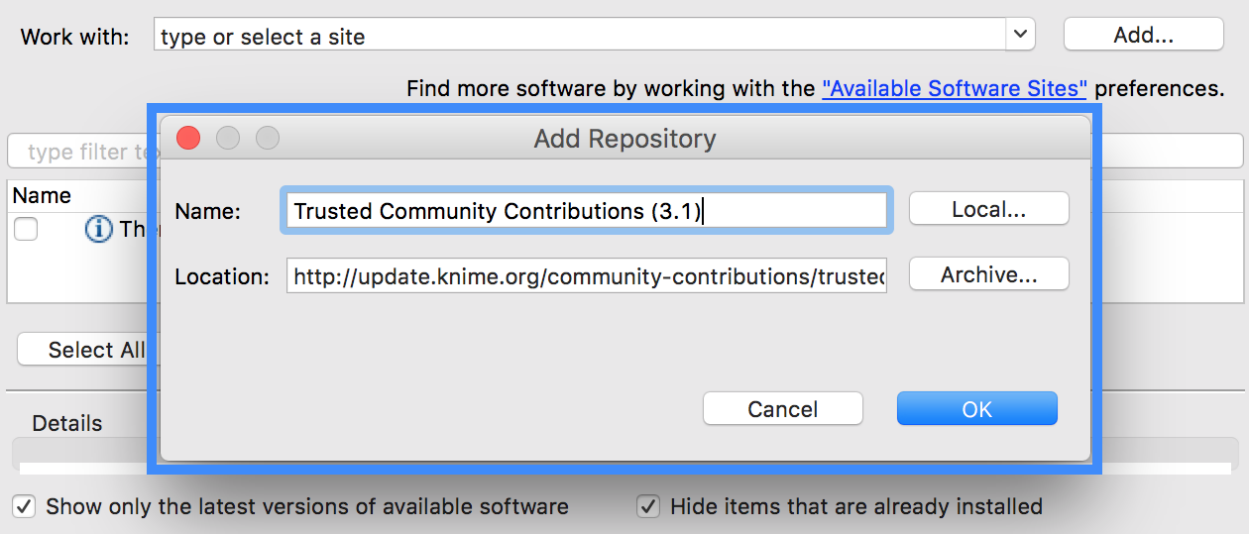
In the opening dialog fill-in the following Information:

Name `Trusted Community Contributions (3.1)`

Location `http://update.knime.org/community-contributions/trusted/3.1`


Install
Available Software
Select a site or enter the location of a site.
Work with:
type or select a site
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
There is no site selected.
Select All
Deselect All
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Cancel
Finish

If you are, by chance, still using an older KNIME version and you do not want to update to the latest version you can find the corresponding update site location at the community-contributions page of the KNIME website.



After pressing OK, KNIME will show you all the contents of the added Update Site, containing also the SeqAn nodes.

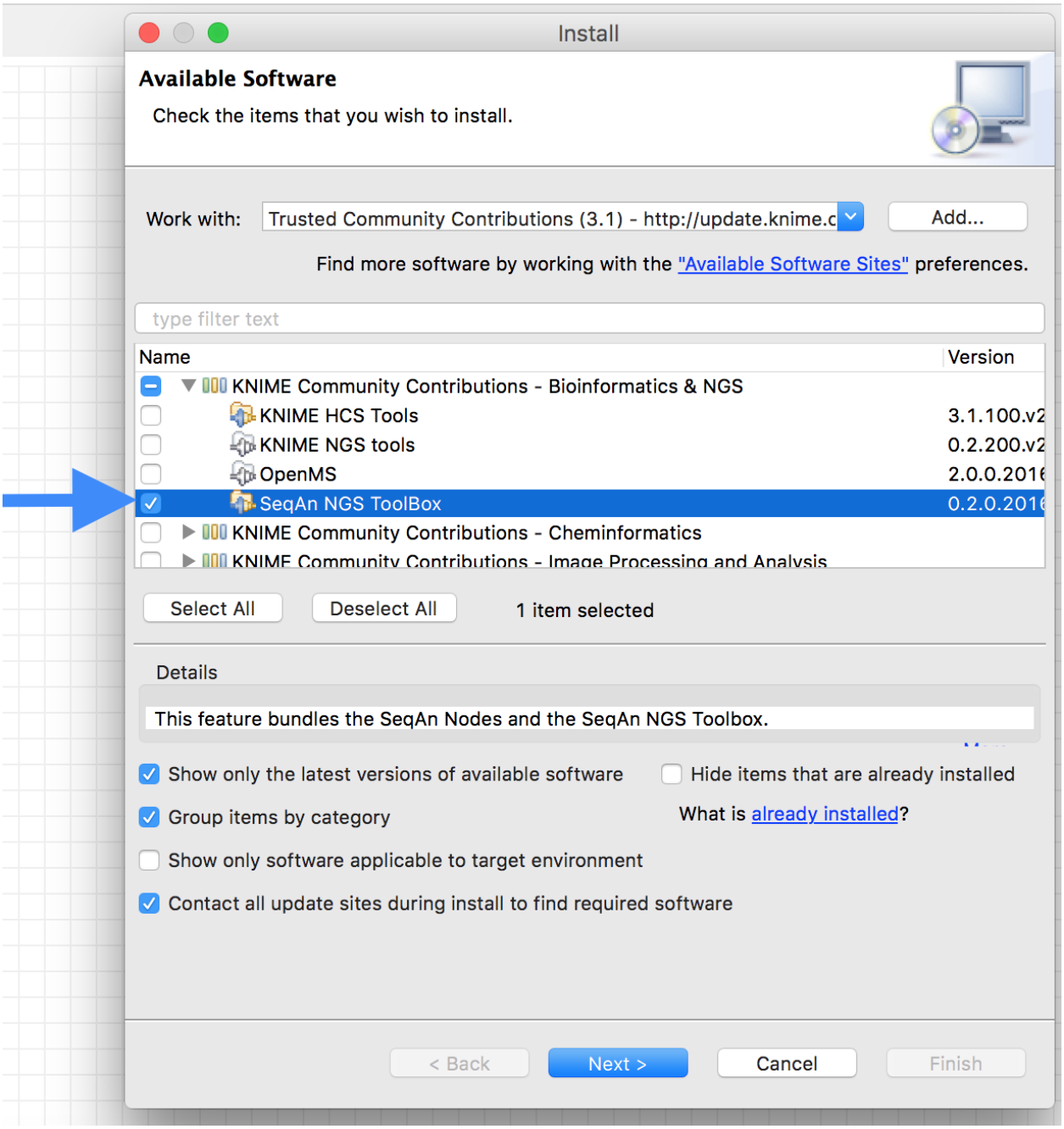
Select the SeqAn NGS Toolbox and click Next. Follow the instructions. After the installation is don KNIME will prompt you to restart. Click OK and KNIME will restart with the newly installed SeqAn nodes will be available under `Community Nodes` category. The installation also includes GenericKnimeNodes which are very useful for using SeqAn nodes in KNIME. This includes file input/output nodes.

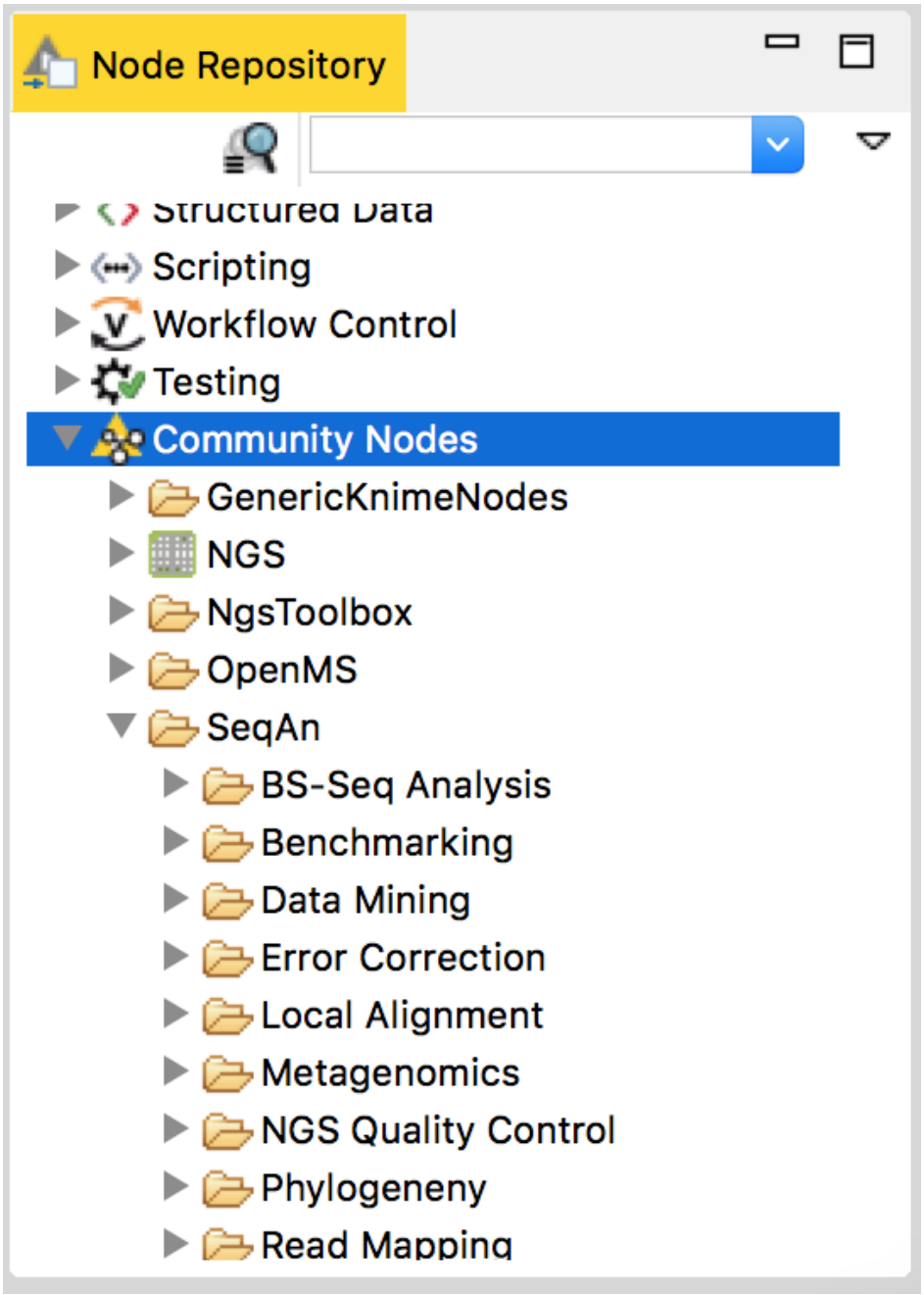
Now you can drag and drop the installed SeqAn nodes to make your desired workflow together with the other KNIME nodes.

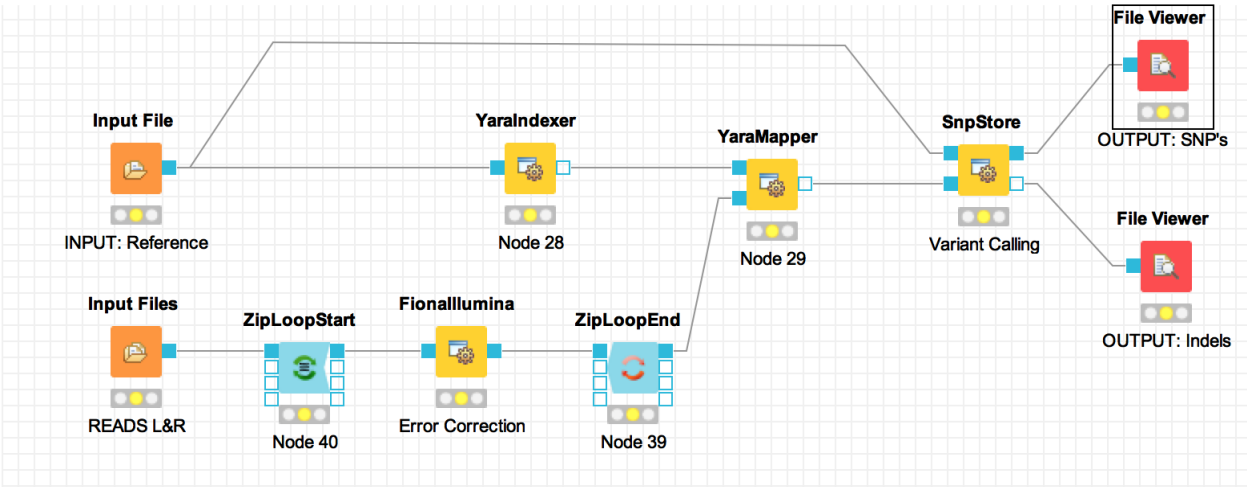
A variant calling workflow (An example)

In this example we will use a read mapper (yara) to map short reads against a reference genome. Then we will use SnpStore to call variants and store the variants as `vcf` and `gff` files. We will also do error correction of Illumina reads before we map them to the reference. In this way we will we can identify SNP's more clearly.

1. Download this zipped `example data` and extract it somewhere appropriate. It contains three files. The file `NC_008253_1K.fa` is a small toy reference genome. Files `sim_reads_l.fq` and `sim_reads_r.fq` are short sequencing paired reads. For each read in one file its mate is contained in the other file.
2. On the left side of the opened KNIME window under KNIME Explorer right click on `LOCAL (Local Workspace)` and chose the menu item `New KNIME Workflow`. You will be presented with a dialog to enter the name and location of the workflow to be created. Give your workflow an approprate name, perhaps something like 'Variant Calling Workflow', and click finish.
3. Drag and drop the nodes shown in the following picture from the `Node Repository` panel on the left bottom side of the KNIME window and arrange/connect them as they are shown in the picture bellow. You can also rename the node from nodeXX to a meaningful name like `INPUT: Reference`. The node name it the text bellow the node. The Node type, which is displayed above the node, cannot be edited.
4. Now it's time to configure our nodes. To configure a node just double-click on it. A configuration dialog will pop up. Let us configure our nodes on our workflow one by one.


Install
Available Software
Check the items that you wish to install.
Work with:
Trusted Community Contributions (3.1) - http://update.knime.c
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
KNIME Community Contributions - Bioinformatics & NGS
KNIME HCS Tools
3.1.100.v2
KNIME NGS tools
0.2.200.v2
OpenMS
2.0.0.2016
SeqAn NGS ToolBox
0.2.0.2016
KNIME Community Contributions - Cheminformatics
KNIME Community Contributions - Image Processing and Analysis
Select All
Deselect All
1 item selected
Details
This feature bundles the SeqAn Nodes and the SeqAn NGS Toolbox.
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Cancel
Finish


Node Repository
Structured Data
Scripting
Workflow Control
Testing
Community Nodes
GenericKnimeNodes
NGS
NgsToolbox
OpenMS
SeqAn
BS-Seq Analysis
Benchmarking
Data Mining
Error Correction
Local Alignment
Metagenomics
NGS Quality Control
Phylogeneny
Read Mapping



a. InputFile Node (`INPUT: Reference`):

- browse and select the file `NC_008253_1K.fa` under Selected file field.
- click OK.

b. InputFiles Node (`READS L&R`):

- click `add` and select both `sim_reads_l.fq` and `sim_reads_r.fq` files.
- click OK.

c. FionaIllumina Node (`Error Correction`):

- set genome-length to 1000

d. SnpStore Node (`Variant Calling`):

- set only-successful-candidates to true.

5. Run the workflow. Right-click on the File Viewer (`OUTPUT: SNP's`) node at the right end of our configured workflow and choose Execute from the menu. As the preceding nodes execute they change their indicator color from yellow to green. When the last node finishes executing do the same to execute the File Viewer (`OUTPUT: indels`)
6. See the results. You can take a look at the results (SNPs/IndDels) by Right-clicking on the corresponding File Viewer node and choose `View: (data view)` from the menu.

Congratulations you have just created a working KNIME workflow using SeqAn nodes!

Use existing workflows and contribute new ones

The git repository https://github.com/seqan/knime_seqan_workflows has quite few workflows ready to run. each workflow is contained in a directory. The directory for a workflow contains an example data and a README file in it. This makes it easier to download and execute the workflow. You can either clone the repository or download individual workflows and execute them with the data provided or with your own data.

With the steps described above you will be able to set up your own workflows in KNIME. If you want to contribute a workflow to the SeqAn community you are encouraged to do so. You can do it as follows:

- Simply clone the workflow git repository into your own github repository and add a new folder `WORKFLOWNAME_workflow`.
- In KNIME export your workflow without the data files as a `.zip` file into that folder.
- Provide a README, a screenshot and some example input data as well.

To get a more clear idea just take a look at the existing workflow folders.

After everything is ready, add...commit and push the new folder into your github repository and make a github pull request to the original workflow repository (https://github.com/seqan/knime_seqan_workflows) and - voila - it will be shared with the community.

ToC

Contents

Generating KNIME Nodes

With the help of the GenericWorkflowNodes one can make his/her command line applications available in KNIME as nodes that can be included as parts of a bigger workflow. In this tutorial you will learn how to make your command line application available as a KNIME node.

Prerequisites

** Eclipse KNIME SDK**
: You can download it from the KNIME Download Site (at the end of the page). We will use Version 3.1.

git
: For Downloading the latest samtools and GenericKnimeNodes.

Apache Ant
: The Generic KNIME Plugins project uses Apache Ant as the build system. On Linux and Mac, you should be able to install it through your package manager. For Windows, see the Apache Ant Downloads (note that samtools does not work on Windows so you will not be able to follow through with this tutorial on Windows).

Overview

KNIME nodes are shipped as Eclipse plugins. The **GenericKnimeNodes** (GWN) package provides the infrastructure to automatically generate such nodes from the description of their command line. The description of the command

line is kept in XML files called *Common Tool Descriptor (CTD) files*.

> **Tip:** *For SeqAn Apps*, thanks to the seqan::ArgumentParser class generating a plugin directory is completely automatic if you are using the SeqAn build infrastructure. All you have to do is run make with target prepare_workflow_plugin. Read the tutorial *Generating SeqAn KNIME Nodes* for more details.

The input of the GWN package is a directory tree with the following structure:

```
plugin_dir
  |
  - plugin.properties
  |
  - descriptors (place your ctd files and mime.types here)
      - app1_name.ctd
      - app2_name.ctd
      - ...
      - mime.types
  |
  - payload (place your binaries here)
  |
  - icons (the icons to be used must be here)
  |
  - DESCRIPTION (A short description of the project)
  |
  - LICENSE (Licensing information of the project)
  |
  - COPYRIGHT (Copyright information of the project)
```

The file plugin.properties

This file contains the plugin configuration. Look at the following `plugin.properties` file as an example:

```
# the package of the plugin
pluginPackage=net.sf.samtools

# the name of the plugin
pluginName=SamTools

# the version of the plugin
pluginVersion=0.1.17

# the path (starting from KNIMEs Community Nodes node)
nodeRepositoyRoot=community

executor=com.genericworkflownodes.knime.execution.impl.LocalToolExecutor
commandGenerator=com.genericworkflownodes.knime.execution.impl.CLICommandGenerator
```

When creating your own plugin directory, you only have to update the first three properties:

pluginPackage
: A Java package path to use for the Eclipse package.

pluginName
: A CamelCase name of the plugin.

pluginVersion
Version of the Eclipse plugin.

The Directory descriptors

The descriptors directory contains two types of files i.e. the *CTD files* for each application in our plugin and a *mime.types* file.

MIME Types

mime.types file is a text file that contains a mapping between MIME types and file extensions. Every file extension, to be either used or produced by the applications in the plugin, has to be registered here. Each line contains the definition of a MIME type. The name of the mime type is followed (separated by a space) by the file extensions associated with the file type. The following example shows how the content of a *mime.types* file looks like.

```
application/x-fasta fa fasta
application/x-fastq fq fastq
application/x-sam sam
application/x-bam bam
```

Important: There may be no ambiguous mappings, like giving a single extension for both *application/x-fasta* and *application/x-fastq* in the example shown above. An extension should be mapped to a single application.

The CTD (Common Tool descriptor) files

For every application with the name `app_name`, there is one CTD file called *${app_name}.ctd*. These file contain the command line description of an application in XML format. They also decide which command-line arguments will be input/output ports or configuration entries in the node to be generated.

Tip: For application developed in SeqAn or applications using the seqan::ArgumentParser for parsing their command-line arguments a CTD file can be generated using a hidden parameter `-write-ctd`.

for example:

```
./seqan_app_name -write-ctd seqan_app_name.ctd
```

Look at the tutorials *Make Your SeqAn App KNIME Ready* and *Generating SeqAn KNIME Nodes* .

Below is an example of a CTD file for SortBam tool for sorting BAM files.

```
<?xml version="1.0" encoding="UTF-8"?>
<tool name="SortBam" version="0.1.17" category="SAM and BAM"
      docurl="http://samtools.sourceforge.net/samtools.shtml">
    <executableName>samtools</executableName>
    <description><![CDATA[SAMtools BAM Sorting.]]></description>
    <manual><![CDATA[samtools sort]]></manual>
    <docurl>Direct links in docs</docurl>
    <cli>
        <clielement optionIdentifier="sort" isList="false" />
        <clielement optionIdentifier="-f" isList="false" />
```

```
        <!-- Following clielements are arguments. You should consider
             providing a help text to ease understanding. -->
        <clielement optionIdentifier="" isList="false">
            <mapping referenceName="bam_to_sam.argument-0" />
        </clielement>
        <clielement optionIdentifier="" isList="false">
            <mapping referenceName="bam_to_sam.argument-1" />
        </clielement>

        <clielement optionIdentifier="" isList="false">
            <mapping referenceName="bam_to_sam.argument-2" />
        </clielement>
    </cli>
    <PARAMETERS version="1.4"
                xsi:noNamespaceSchemaLocation="http://open-ms.sourceforge.net/schemas/
↪Param_1_4.xsd"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <NODE name="bam_to_sam" description="SAMtools BAM to SAM conversion">
            <ITEM name="argument-0" value="" type="input-file" required="true"
                  description="Input BAM file." supported_formats="*.bam" />
            <ITEM name="argument-1" value="" type="output-file" required="true"
                  description="Output BAM file." supported_formats="*.bam" />
            <ITEM name="argument-2" value="" type="string" required="true"
                  description="Sort by query name (-n) instead of position (default)"␣
↪restrictions=",-n" />
        </NODE>
    </PARAMETERS>
</tool>
```

Hint: If a `<clielement>` does provides an empty `optionIdentifier` then it is a positional argument without a flag (examples for parameters with flags are `-n 1`, `--number 1`).

If a `<clielement>` does not provide a `<mapping>` then it is passed regardless of whether has been configured or not.

The `samtools_sort_bam` tool from above does not provide any configurable options but only two arguments. These are by convention called `argument-0` and `argument-1` but could have any name.

Also, we always call the program with `view -f` as the first two command line arguments since we do not provide a mapping for these arguments.

Click **more...** to see the description of the tags and the attributes in the CTD XML file:

CTD Tags attributes:

/tool
: The root tag.

/tool@name
: The CamelCase name of the tool as shown in KNIME and part of the class name.

/tool@version
: The version of the tool.

/toll@category
: The path to the tool's category.

/tool/executableName
The name of the executable in the payload ZIP's *bin* dir.

/tool/description
Description of the tool.

/tool/manual
Long description for the tool.

/tool/docurl
URL to the tool's documentation.

/tool/cli
Container for the `<clielement>` tags. These tags describe the command line options and arguments of the tool. The command line options and arguments can be mapped to parameters which are configurable through the UI. The parameters are stored in **tool/PARAMETERS**

/tool/cli/clielement
There is one entry for each command line argument and option.

/tool/cli/clielement@optionIdentifier
The identifier of the option on the command line. For example, for the `-l` option of `ls`, this is `-l`.

/tool/cli/clielement@isList
Whether or not the parameter is a list and multiple values are possible. One of `true` and `false`.

/tool/cli/clielement/mapping
Provides the mapping between a CLI element and a PARAMETER.

/tool/cli/clielement/mapping@referenceName
The path of the parameter. The parameters `<ITEM>` s in **tool/PARAMETERS** are stored in nested `<NODE>` tags and this gives the path to the specific parameter.

/tool/PARAMETERS
Container for the `<NODE>` and `<ITEM>` tags. The `<PARAMETERS>` tag is in a diferent namespace and provides its own XSI.

/tool/PARAMETERS@version
Format version of the `<PARAMETERS>` section.

/tool/PARAMETERS/.../NODE
A node in the parameter tree. You can use such nodes to organize the parameters in a hierarchical fashion.

/tool/PARAMETERS/.../NODE@advanced
Boolean that marks an option as advanced.

/tool/PARAMETERS/.../NODE@name
Name of the parameter section.

/tool/PARAMETERS/.../NODE@description
Documentation of the parameter section.

/tool/PARAMETERS/.../ITEM
Description of one command line option or argument.

/tool/PARAMETERS/.../ITEM@name
Name of the option.

/tool/PARAMETERS/.../ITEM@value
Default value of the option. When a default value is given, it is passed to the program, regardless of whether the user touched the default value or not.

/tool/PARAMETERS/.../ITEM@type
Type of the parameter. Can be one of `string`, `int`, `double`, `input-file`, `output-path`, `input-prefix`, or `output-prefix`. Booleans are encoded as `string` with the `restrictions` attribute set to `"true,false"`.

/tool/PARAMETERS/.../ITEM@required

Boolean that states whether the parameter is required or not.

/tool/PARAMETERS/.../ITEM@description

Documentation for the user.

/tool/PARAMETERS/.../ITEM@supported_formats

A list of supported file formats. Example: `"*.bam,*.sam"`.

/tool/PARAMETERS/.../ITEM@restrictions

In case of `int` or `double` types, the restrictions have the form `min:`, `:max`, `min:max` and give the smallest and/or largest number a value can have. In the case of `string` types, restrictions gives the list of allowed values, e.g. `one,two,three`. If the type is `string` and the restriction field equals `"true,false"`, then the parameter is a boolean and set in case `true` is selected in the GUI. A good example for this would be the `-l` flag of the `ls` program.

The Directory payload

The directory `payload` contains ZIP files with the executable tool binaries. Usually there is one ZIP file for each platform `(Linux, Windows, and Mac Os X)` architecture `(32Bit or 64Bit)``combination. The names of the files are ``binaries_${plat}_${arch}.zip` where `${plat}` is one of `lnx`, `win`, or `mac`, and `${arch}` is one of `32` and `64`. In this way the appropriate binaries will be used based on the system that KNIME is running on. Some, even all, of the zip files representing different architectures can be missing. That means the node will not be functional on a KNIME instance installed on the architecture(s) corresponding to missing zip file(s). Nevertheless the payload directory has to be present even if it is empty.

Each ZIP file contains a directory `/bin` which is used as the search path for the binary given by `<executableName>` and an INI file `/binaries.ini` which can be used to set environment variables before executing any of tools.

The Directory icons

Here we put icons for different purposes.

- An image file with a name *category.png* (15x15 px): an icon for categories in the KNIME node explorer tree.
- An image file with a name *splash.png* (50x50 px): an icon to be displayed in the KNIME splash screen.
- An image file with a name *app_name.png* (15x15 px): an icon to be displayed with the corresponding node of the application `app_name`. This is done once for each app but it's optional.

The files DESCRIPTION, LICENSE, COPYRIGHT

- **DESCRIPTION**: A text file with your project's description.
- **LICENSE**: A file with the license of the project.
- **COPYRIGHT**: A file with copyright information for the project.

The GWN project provides tools to convert such a plugin directory into an Eclipse plugin. This plugin can then be launched together with KNIME. The following picture illustrates the process.

Running Example

We will adapt some functions from the samtools package to KNIME:

BamToSam
: This tool will execute `samtools view -o ${OUT} ${IN}`.

SamToBam
: This tool will execute `samtools view -Sb -o ${OUT} ${IN}`.

SortBam
: This tool will execute `samtools sort -o ${OUT} ${IN}`.

Preparation: Building samtools and Downloading GenericKnimeNodes

We will work in a new directory *knime_samtools* (we will assume that the directory is directly in your *$HOME* for the rest of the tutorial. First we need to download samtools-1.3 from http://www.htslib.org/download/. Alternatively you can also use this direct link or use either of *wget* or *curl* utilities from your command line as follows.

wget

```
knime_samtools # wget https://github.com/samtools/samtools/releases/download/1.3/
↪samtools-1.3.tar.bz2
```

curl

```
knime_samtools # curl -OL https://github.com/samtools/samtools/releases/download/1.3/
↪samtools-1.3.tar.bz2
```

Now let us extract and build samtools

```
knime_samtools # tar -jxvf samtools-1.3.tar.bz2
...
knime_samtools # cd samtools-1.3
samtools-1.3 # ./configure
samtools-1.3 # make
...
samtools # ls -l samtools
-rwxr-xr-x 1 user group 1952339 May  7 16:36 samtools
samtools # cd ..
knime_samtools #
```

Then, we need to download GenericKnimeNodes:

```
knime_samtools # git clone git://github.com/genericworkflownodes/GenericKnimeNodes.git
```

Preparation: Installing KNIME File Handling

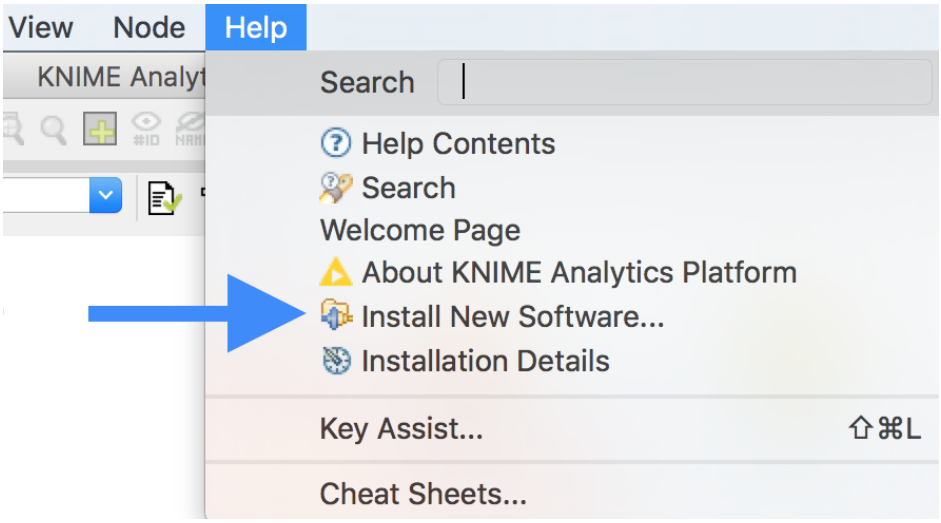
We need to install support for file handling nodes in KNIME. In order to do that - Launch your Eclipse-KNIME-SDK

Tip: Launching Eclipse-KNIME-SDK

- If you are on a linux system you should browse to your knime_eclipse_3.1 installation double click on `eclipse` executable.

- If you are using MacOS X then you can launch the KNIME SDK 3.1 as you would launch any other application.

- Open the window for installing Eclipse plugins; in the program's main menu: `Help > Install New Software...`.



- On the install window enter `http://www.knime.org/update/3.1` into the `Work with:` field, enter `file` into the search box, and finally select `KNIME File Handling Nodes` in the list.
- Then, click `Next` and follow through with the installation of the plugin. When done, Eclipse must be restarted.; in the program's main menu: `Help > Install New Software...`.

Obtaining the Demo Workflow Plugin Directory

Please download the file `workflow_plugin_dir.zip` to your `knime_samtools` directory and look around in the archive. Also have a look into `binaries_*_*.zip` files in *payload*. The structure of this ZIP file is similar to the one explained in the Overview section at the beginning of this tutorial.

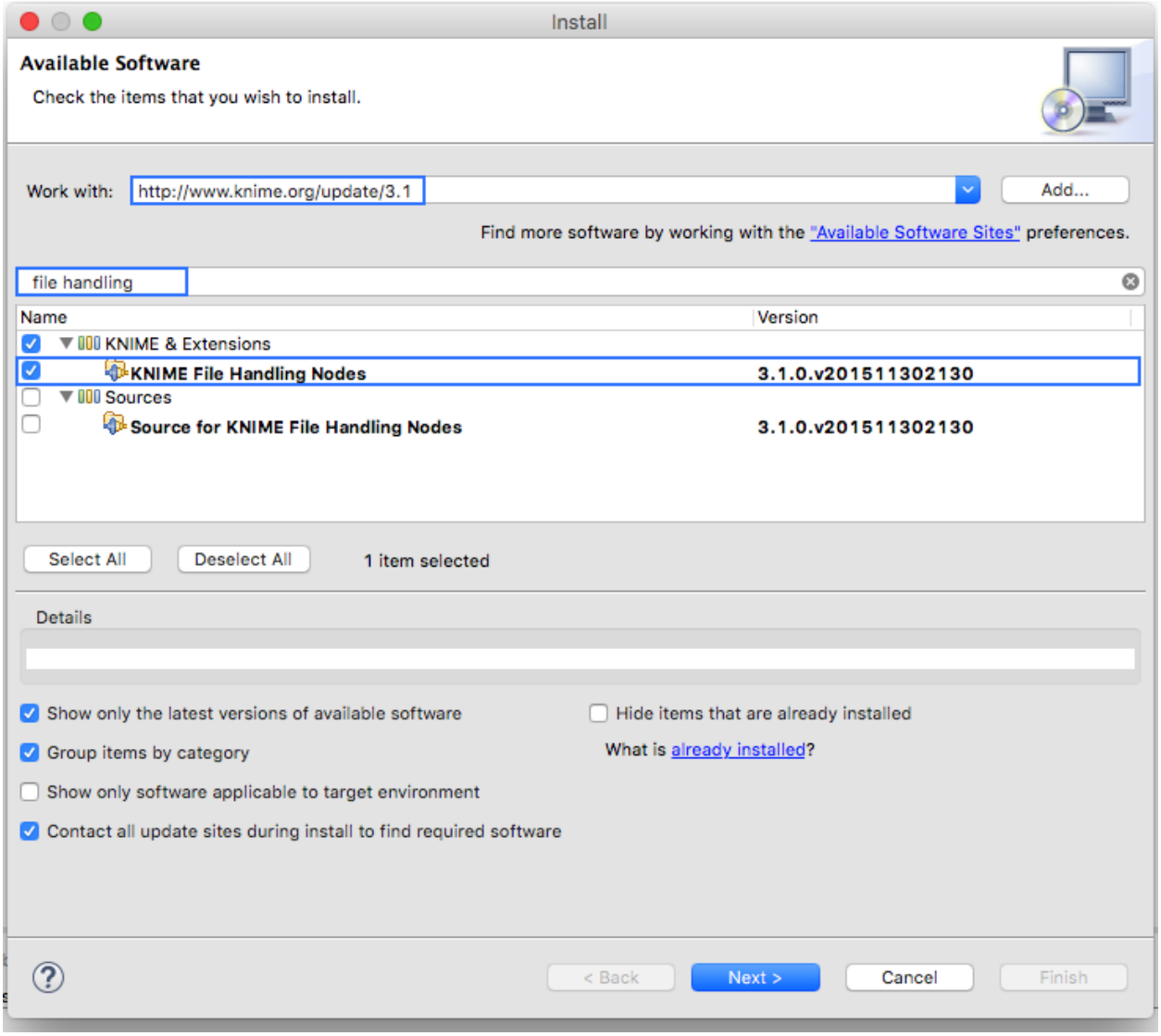
Creating an Exclipse Plugin from the Plugin Directory

The next step is to use GKN to create an Eclipse plugin from the workflow plugin directory. For this, change to the directory GenericKnimeNodes that we cloned using git earlier. We then execute ant and pass the variables *knime.sdk* with the path to the KNIME SDK that you downloaded earlier and *plugin.dir* with the path of our extracted `workflow_plugin_dir` directory.

> **Tip: Path to Eclipse-KNIME-SDK** - If you are on a linux system the path to *knime.sdk* is the path to your `knime_eclipse_3.1.0` - If you are using MacOS X then the default path to *knime.sdk* is `/Applications/KNIME\ SDK\ 3.1.0.app/`

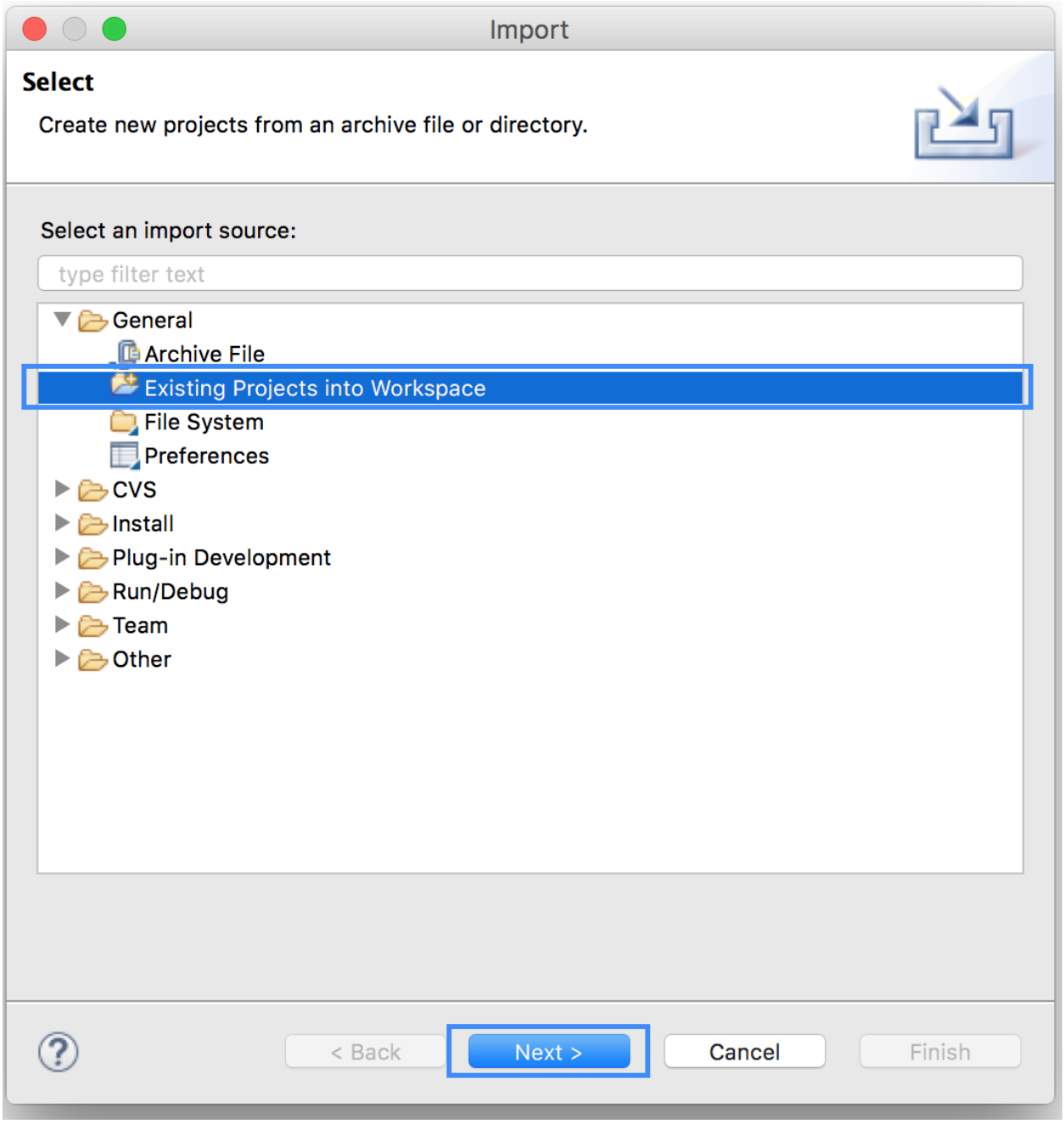
```
knime_samtools # cd GenericKnimeNodes
GenericKnimeNodes # ant -Dknime.sdk=${HOME}/eclipse_knime_3.1 \
                        -Dplugin.dir=$HOME/knime_samtools/workflow_plugin_dir
```

This generates an Eclipse plugin with wrapper classes for our nodes. The generated files are within the *generated_plugin* directory of the directory *GenericKnimeNodes*.


Install
Available Software
Check the items that you wish to install.
Work with:
http://www.knime.org/update/3.1
Add...
Find more software by working with the "Available Software Sites" preferences.
file handling
Name
Version
KNIME & Extensions
KNIME File Handling Nodes
3.1.0.v201511302130
Sources
Source for KNIME File Handling Nodes
3.1.0.v201511302130
Select All
Deselect All
1 item selected
Details
Show only the latest versions of available software
Group items by category
Show only software applicable to target environment
Contact all update sites during install to find required software
Hide items that are already installed
What is already installed?
< Back
Next >
Cancel
Finish

Importing the Generated Projects into Eclipse

On the KNIME Eclipse SDK window go to the menu `File > Import...`. In the `Import` window, select `General > Existing Project Into Workspace`

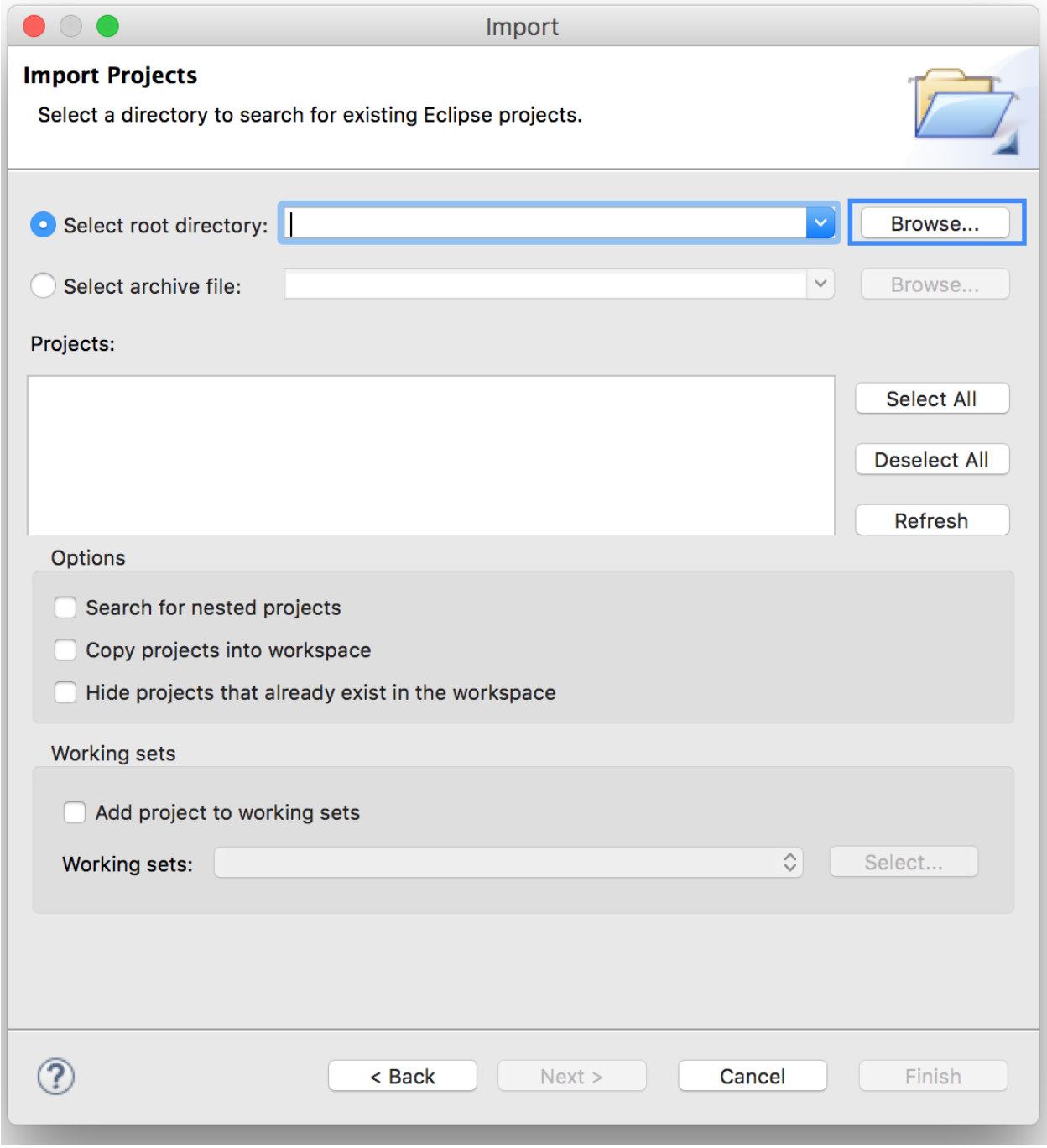


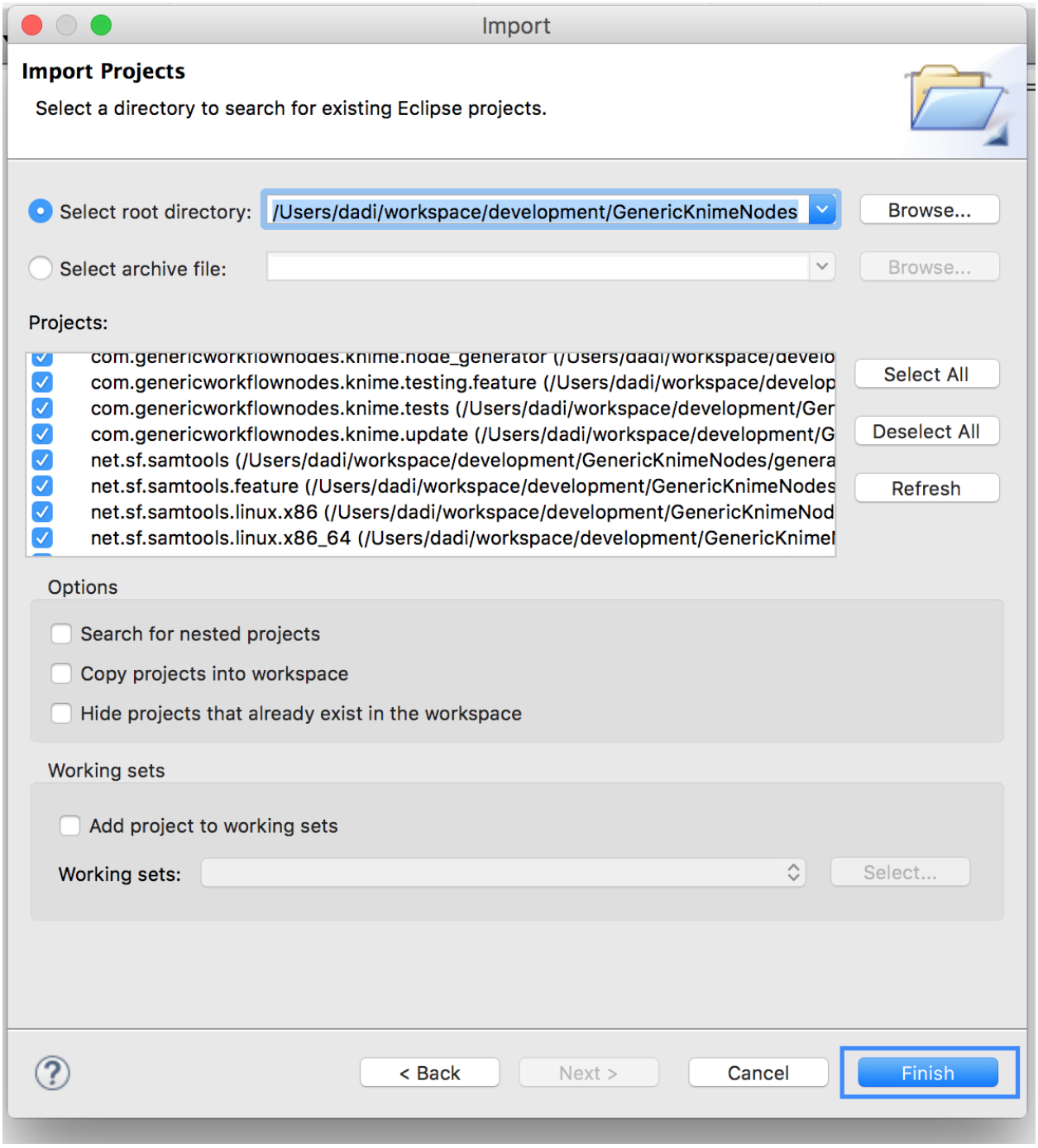
On the pop-up dialog, click `Browse...` next to `Select root directory`.

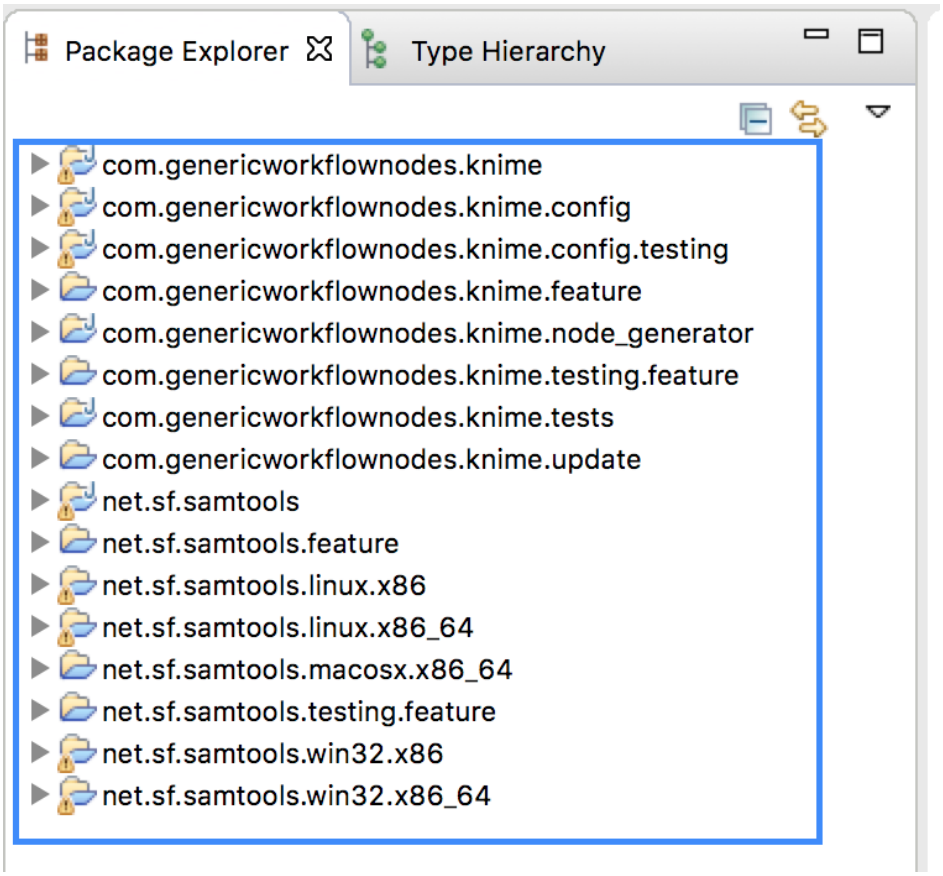
Then, select the directory of your "GenericWorkflowNodes" checkout. The final dialog should then look as follows.

Clicking finish will import (1) the GKN classes themselves and (2) your generated plugin's classes.

Now, the packages of the GKN classes and your plugin show up in the left `Package Explorer` pane of Eclipse.


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Browse...
Select archive file:
Browse...
Projects:
Select All
Deselect All
Refresh
Options
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Cancel
Finish


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
/Users/dadi/workspace/development/GenericKnimeNodes
Browse...
Select archive file:
Browse...
Projects:
com.genericworkflownodes.knime.node_generator (/Users/dadi/workspace/develo
com.genericworkflownodes.knime.testing.feature (/Users/dadi/workspace/develop
com.genericworkflownodes.knime.tests (/Users/dadi/workspace/development/Ger
com.genericworkflownodes.knime.update (/Users/dadi/workspace/development/G
net.sf.samtools (/Users/dadi/workspace/development/GenericKnimeNodes/genera
net.sf.samtools.feature (/Users/dadi/workspace/development/GenericKnimeNodes
net.sf.samtools.linux.x86 (/Users/dadi/workspace/development/GenericKnimeNod
net.sf.samtools.linux.x86_64 (/Users/dadi/workspace/development/GenericKnimeI
Select All
Deselect All
Refresh
Options
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Cancel
Finish


Package Explorer
Type Hierarchy
com.genericworkflownodes.knime
com.genericworkflownodes.knime.config
com.genericworkflownodes.knime.config.testing
com.genericworkflownodes.knime.feature
com.genericworkflownodes.knime.node_generator
com.genericworkflownodes.knime.testing.feature
com.genericworkflownodes.knime.tests
com.genericworkflownodes.knime.update
net.sf.samtools
net.sf.samtools.feature
net.sf.samtools.linux.x86
net.sf.samtools.linux.x86_64
net.sf.samtools.macosx.x86_64
net.sf.samtools.testing.feature
net.sf.samtools.win32.x86
net.sf.samtools.win32.x86_64

Hint: Information: Synchronizing `ant` build result with Eclipse.

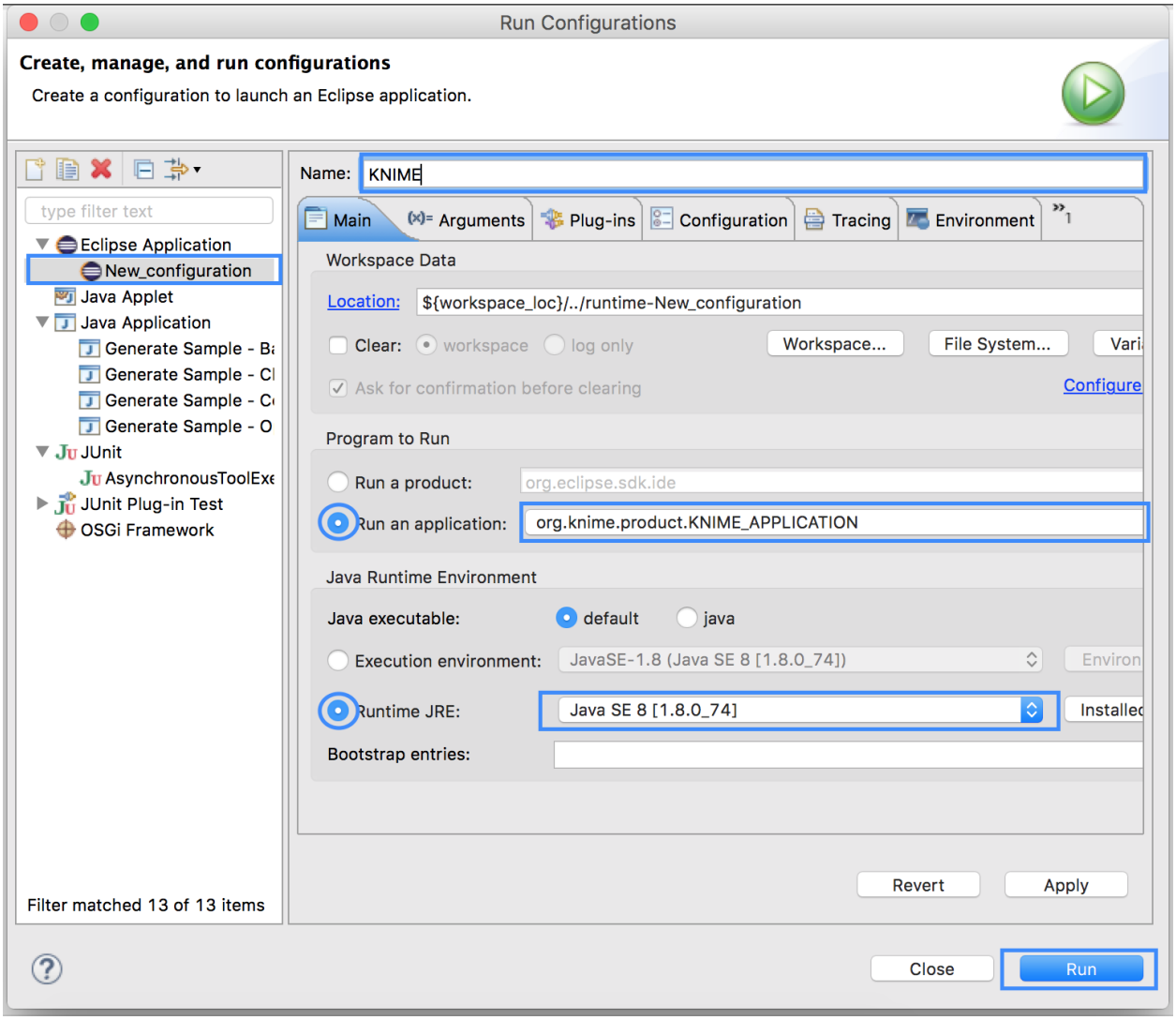
Since the code generation happens outside of Eclipse, there are often problems caused by Eclipse not recognizing updates in generated *java* files. After each call to `ant`, you should clean all built files in all projects by selecting the menu entries `Project > Clean...`, selecting `Clean all projects`, and then clicking `OK`.

Then, select all projects in the `Package Explorer`, right-click and select `Refresh`.

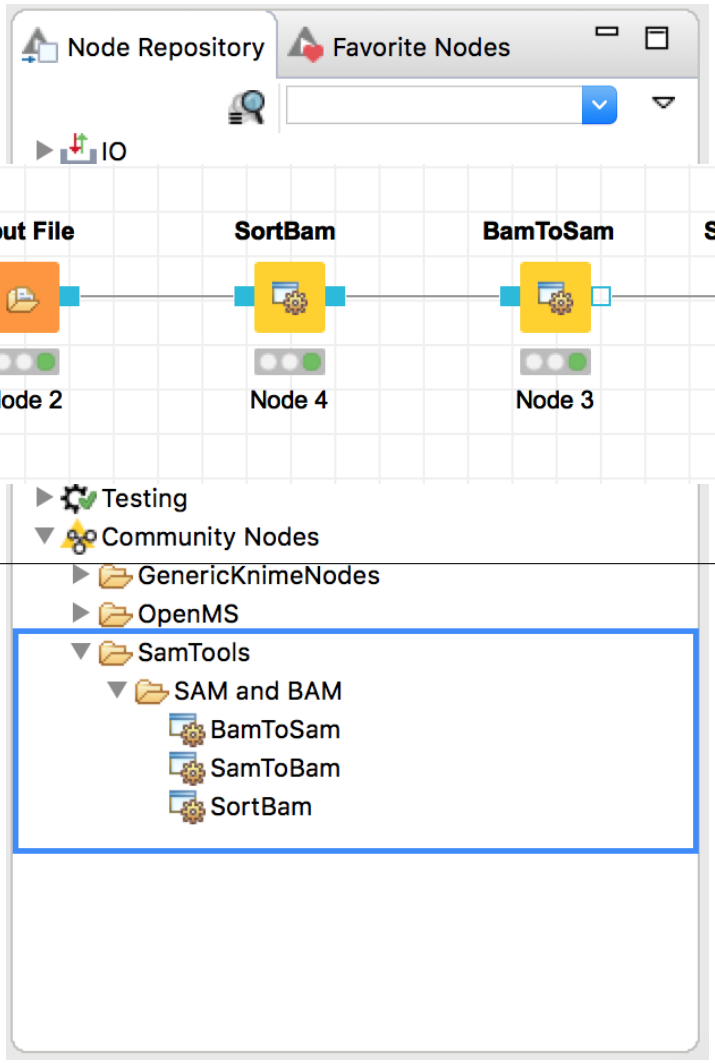
Launching Eclipse with our Nodes

Finally, we have to launch KNIME with our plugin. We have to create a run configuration for this. Select `Run > Run Configurations...`.

In the `Run Configurations` window, select `Eclipse Application` on the left, then create the small `New launch configuration` icon on the top left (both marked in the following screenshot). Now, set the `Name` field to "KNIME", select `Run an application` and select `org.knime.product.KNIME_APPLICATION` in the drop down menu. Finally, click `Run`.



Your tool will show up in the tool selector in `community/SAM and BAM`.



Here is an example KNIME workflow with the nodes that we just created.

ToC

Contents

Make Your SeqAn App KNIME Ready

Learning Objective

You will learn how to use the the seqan::ArgumentParser

and the SeqAn build system so that, at the end, a new SeqAn application can be integrated in KNIME easily. After completing this tutorial, you will be able write a new SeqAn application that can be imported into a KNIME Eclipse plugin with a couple of commands.

Difficulty Basic

Duration 1.5 h

Prerequisites *A First Example*, *Parsing Command Line Arguments*

In this tutorial you will learn how to write a SeqAn app, which can be, easly converted into a KNIME node.

The first part consists of preparing a dummy app such that it can be used in a KNIME workflow and in the second part you are asked to adapt the app such that it becomes a simple quality control tool.

Using the seqan::ArgumentParser

When we add options to the parser using addOption, we pass an ArgParseOption object together with the parser. The ArgumentType of this ArgParseOption object is highly correlated to how the node generated from our application will look like.

The ArgumentType can be one of the following

```
*STRING:*  Argument is a string.
*INTEGER:* Argument is a signed 32 bit integer.
*INT64:* Argument is a signed 64 bit integer.
*DOUBLE:* Argument is a floating point number stored as double.
*INPUT_FILE:*  Argument is an input file.
*OUTPUT_FILE:* Argument is an output file.
*INPUT_PREFIX:*  Argument is a prefix to input file(s).
*OUTPUT_PREFIX:* Argument is a prefix to output file(s).
```

Consider the following example application.

```
```

While adding an ArgParseOption to your ArgumentParser you should consider the following points.

- KNIME needs to know the input and output ports of a node. Therefore we must specify them using `ArgParseArgument::INPUT_FILE` or `ArgParseArgument::OUTPUT_FILE` as can be seen in the example above.
- In addition, KNIME needs to know the valid file endings, which you can specify with setValidValues, which is also shown in the example.

Tip: Later, when building workflows, you can only connect an output-port of a node to the input-port of the next one if only they have a compatible file endings.

- There are special types of input/output ports which are prefixes to a list of files. Such ports are specified using `ArgParseArgument::INPUT_PREFIX` or `ArgParseArgument::OUTPUT_PREFIX`. You can only connect an output prefix port to an input prefix port and vise-versa.

Using the SeqAn build system to generate KNIME nodes

If you are using the SeqAn build system you can generate a workflow plugin directory for all the SeqAn apps including your new one using the target `prepare_workflow_plugin`.

In order for your application to turn into a KNIME node, you should register your app `e.g. my_app`, by simply adding the line:

```
set (SEQAN_CTD_EXECUTABLES ${SEQAN_CTD_EXECUTABLES} <my_app> CACHE INTERNAL "")
```

to the end of the *CMakeList.txt* file of your application. All applications with this line in their *CMakeList.txt* file will be included in the generated plugin when building the target‘‘prepare_workflow_plugin‘‘.

Tip: *If You are not using the SeqAn build system for some reason*, but you used the seqan::ArgumentParser as recommended above, you still can generate a CTD file of your application. After building your application and go to the directory containing the executable of your application and run the following.

```
./seqan_app_name -write-ctd seqan_app_name.ctd
```

This will give you the CTD file of your command-line tool. Then you can follow *Overview* section of the tutorial Generating KNIME Nodes to prepare a plugin directory of your application.

ToC

Contents

Generating SeqAn KNIME Nodes

Learning Objective You will learn how to import applications written in SeqAn into the KNIME Eclipse plugin. After completing this tutorial, you will be able to use self-made applications in KNIME workflows.

Difficulty Very basic

Duration 1 h

Prerequisites

Eclipse KNIME SDK You can download it from the KNIME Download Site (at the end of the page). We will use Version 3.1.

git For downloading the latest GenericKnimeNodes.

Apache Ant The Generic KNIME Plugins project uses Apache Ant as the build system. On Linux and Mac, you should be able to install it through your package manager. For Windows, see the Apache Ant Downloads.

Important: The steps described here are necessary if you want to develop and test new SeqAn apps in KNIME. If you only want to use existing SeqAn apps in KNIME follow *Creating Workflows with SeqAn Nodes in KNIME*.

We will generate a simple SeqAn KNIME node from a SeqAn app that reads a fastq file from disk and just writes it back. We start by installing the necessary software. Afterwards, we explain which steps are required in order to prepare a SeqAn app to be used in KNIME, and finally, we show how to import the app into KNIME. The following section provides some more information on the plugin structure and where the necessary information is stored. Note that this tutorial is mainly written for MacOS and Linux users, but Windows users should also be able to follow through.

Preparation: Downloading GenericKnimeNodes

We will work in a new directory *knime_node* (we will assume that the directory is directly in your *$HOME* for the rest of the tutorial).

```
knime_node # git clone git://github.com/genericworkflownodes/GenericKnimeNodes.git
```

Preparation: Installing KNIME File Handling

We need to install support for file handling nodes in KNIME. In order to do that - Launch your Eclipse-KNIME-SDK

Tip: Launching Eclipse-KNIME-SDK

- If you are on a linux system you should browse to your knime_eclipse_3.1 installation double click on `eclipse` executable.
- If you are using MacOS X then you can launch the KNIME SDK 3.1 as you would launch any other application.

- Open the window for installing Eclipse plugins; in the program's main menu: `Help > Install New Software...`.
- On the install window enter `http://www.knime.org/update/3.1` into the `Work with:` field, enter `file` into the search box, and finally select `KNIME File Handling Nodes` in the list.
- Then, click `Next` and follow through with the installation of the plugin. When done, Eclipse must be restarted.; in the program's main menu: `Help > Install New Software...`.

Generating KNIME Nodes for SeqAn Apps

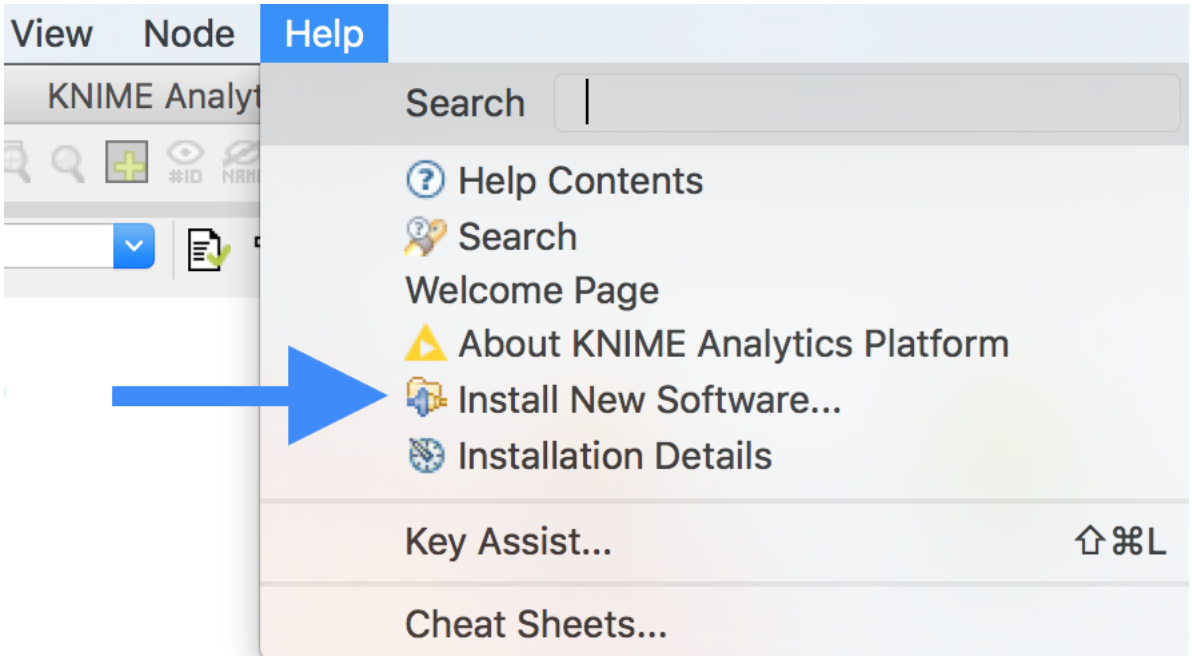
You can generate a workflow plugin directory for the SeqAn apps using the `prepare_workflow_plugin` target.

In order for your application to turn into a KNIME node, you have to add the line:

```
set (SEQAN_CTD_EXECUTABLES ${SEQAN_CTD_EXECUTABLES} <my_app> CACHE INTERNAL "")
```

to the end of the *CMakeList.txt* file of your application.

The following example will demonstrate the creation of a SeqAn app and its registration as a KNIME node.




```
~ # git clone http://github.com/seqan/seqan seqan-src
~ # cd seqan-src
~ # ./util/bin/skel.py app knime_node .
```

Now open the file *seqan-src/apps/knime_node/knime_node.cpp* and replace its content with the one found in *seqan-src/demos/knime_node.cpp*. The code implements the reading of a read file and its storage somewhere on the disk.

In order to register the app `knime_node`, you simply add the line

```
set (SEQAN_CTD_EXECUTABLES ${SEQAN_CTD_EXECUTABLES} knime_node CACHE INTERNAL "")
```

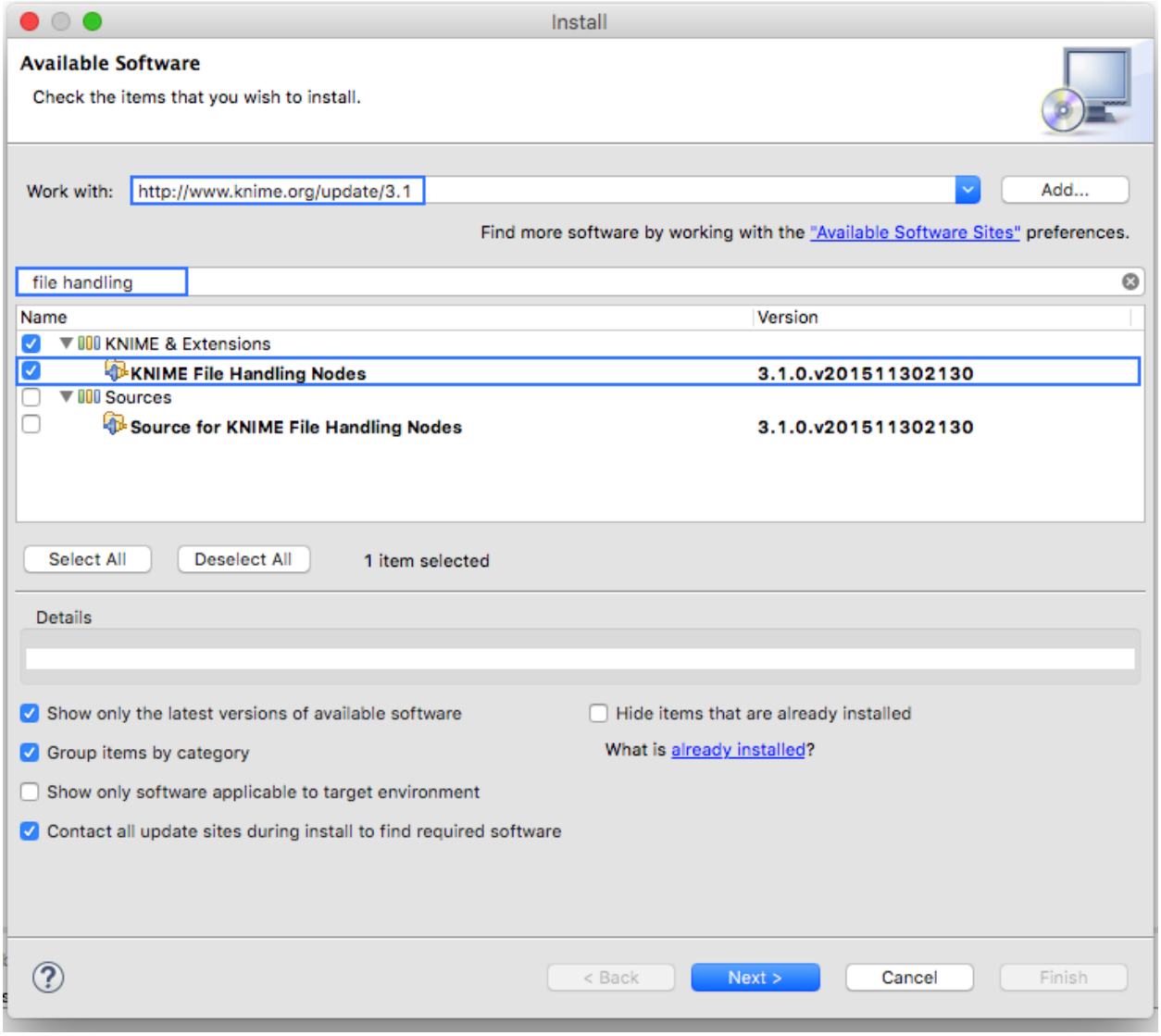
to *seqan-trunk/apps/knime_node/CMakeList.txt*.

Then, you can generate the Knime Nodes/Eclipse plugin. First, change to the directory GenericKnimeNodes that we cloned using git earlier. We then execute ant and pass the variables `knime.sdk` with the path to the KNIME SDK that you downloaded earlier and `plugin.dir` with the path of our plugin directory.

```
~ # mkdir -p seqan-build/release
~ # seqan-build/release
~ # cd seqan-build/release
release # cmake ../../seqan-src
release # make prepare_workflow_plugin
release # cd ~/knime_node/GenericKnimeNodes
GenericKnimeNodes # ant -Dknime.sdk=${HOME}/eclipse_knime_2.8.0 \
                      -Dplugin.dir=${HOME}/seqan-build/release/workflow_plugin_dir
```

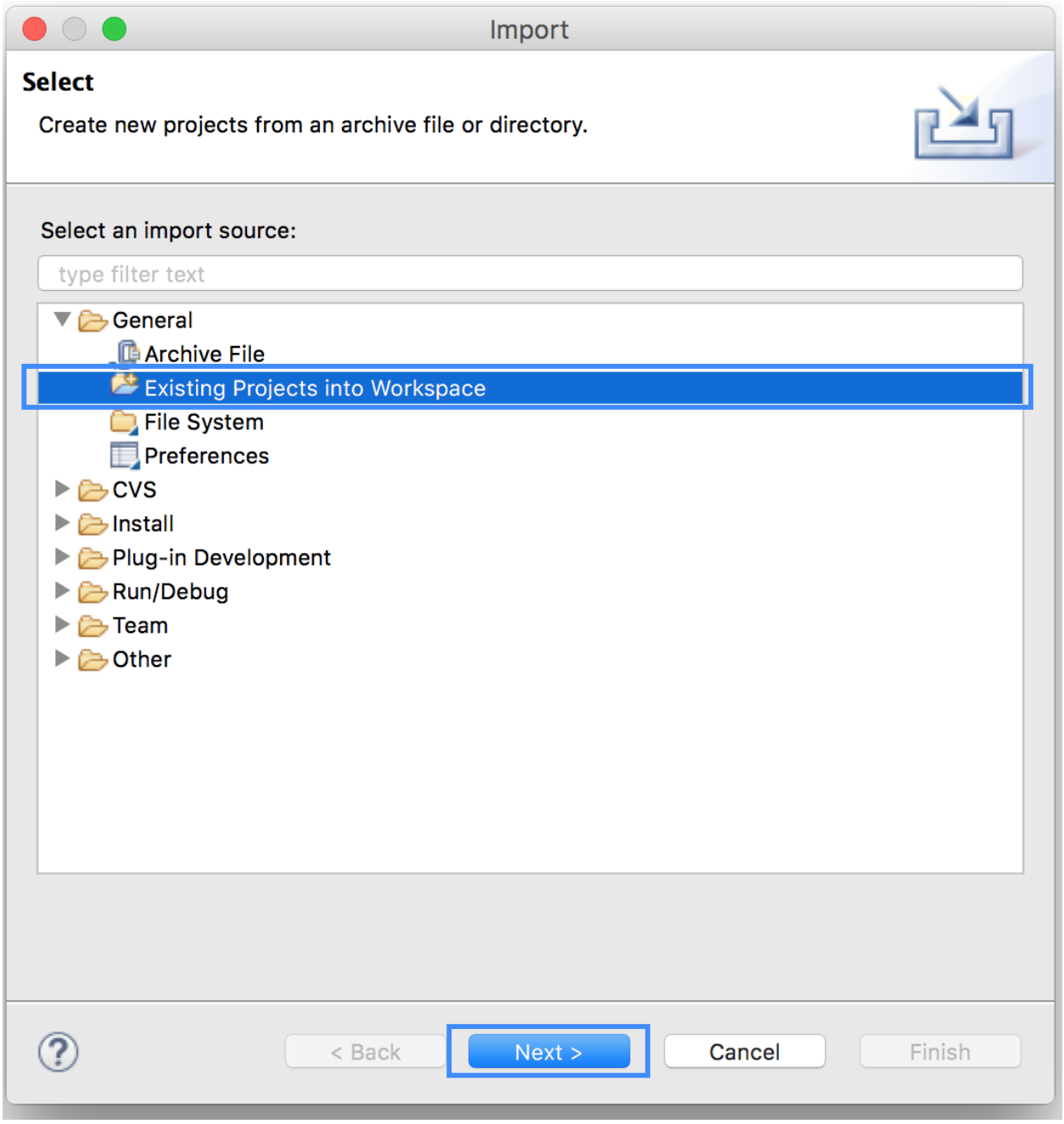
The generated files are within the *generated_plugin* directory of the directory *GenericKnimeNodes*.

If you ran into problems, you may copy the file knime_node_app.zip, which contains the `knime_node` app and the adjusted *CMakeList.txt* file. Unpack this file in the `apps` directory. You still have to call *ant* though.


Install
Available Software
Check the items that you wish to install.
Work with:
http://www.knime.org/update/3.1
Add...
Find more software by working with the "Available Software Sites" preferences.
file handling
Name
Version
KNIME & Extensions
KNIME File Handling Nodes
3.1.0.v201511302130
Sources
Source for KNIME File Handling Nodes
3.1.0.v201511302130
Select All
Deselect All
1 item selected
Details
Show only the latest versions of available software
Group items by category
Show only software applicable to target environment
Contact all update sites during install to find required software
Hide items that are already installed
What is already installed?
< Back
Next >
Cancel
Finish

Importing the Generated Projects into Eclipse

In the main menu, go to `File > Import...`. In the `Import` window, select `General > Existing Project Into Workspace`.

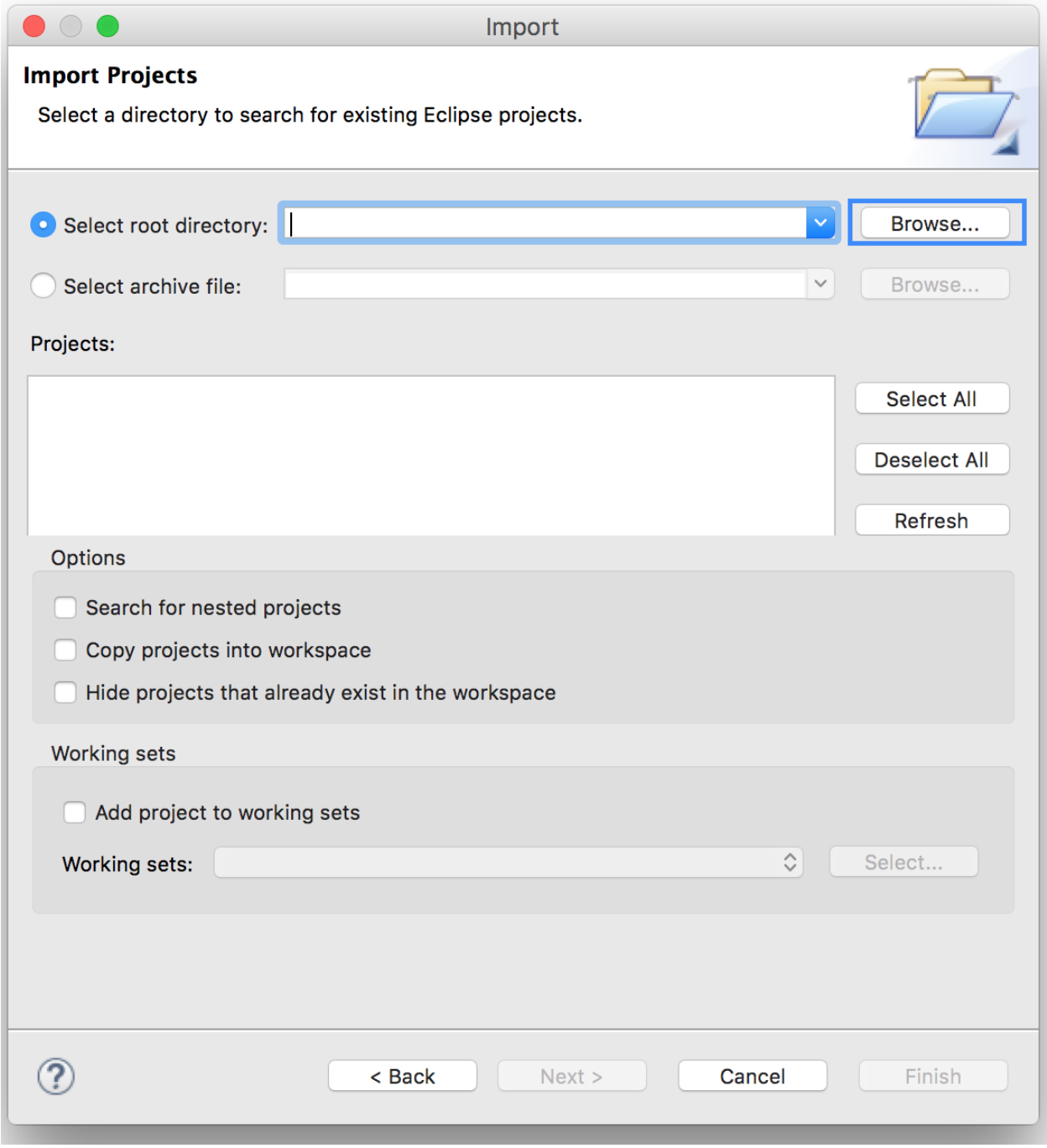


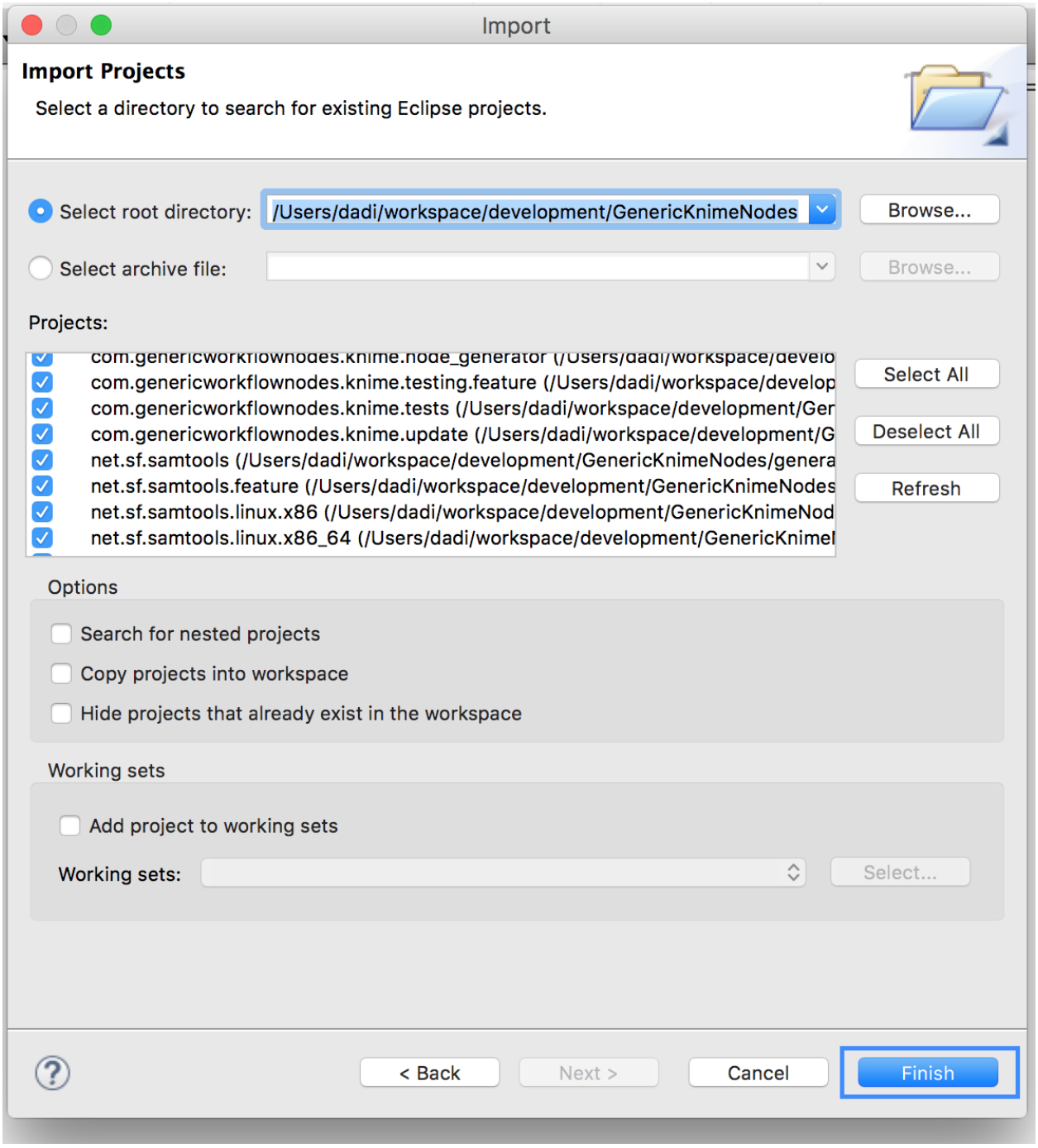
In the next dialog, click `Browse...` next to `Select root directory`.

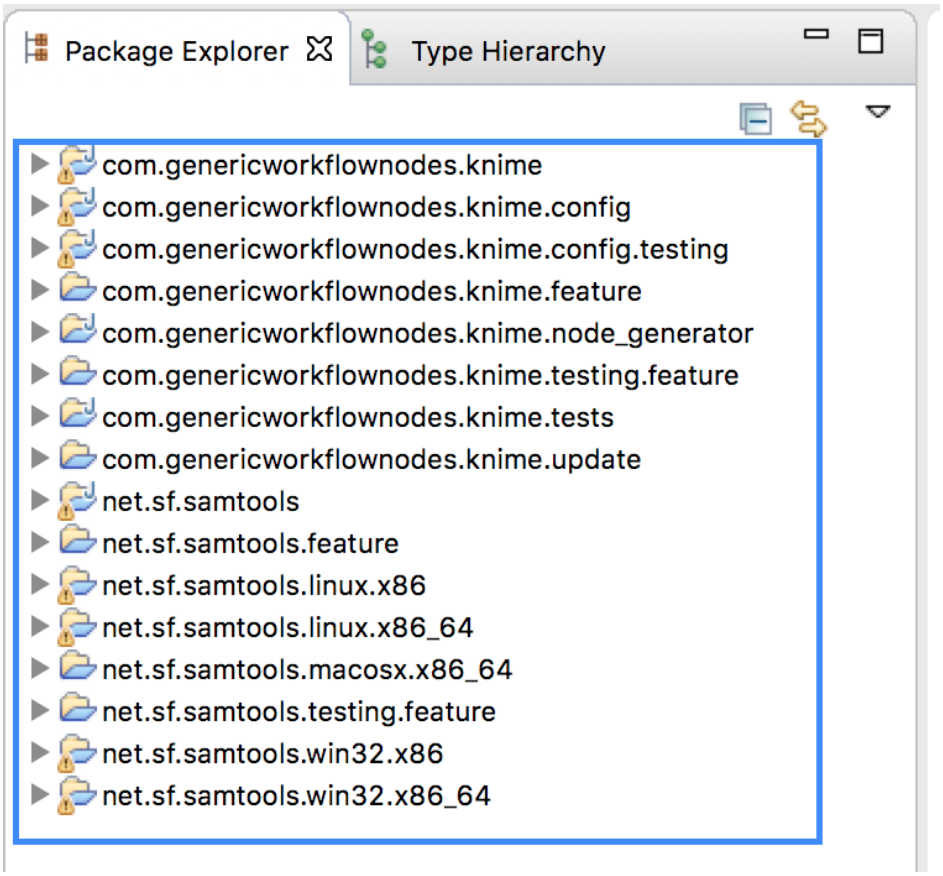
Then, select the directory of your "GenericWorkflowNodes" checkout. The final dialog should then look as follows.

Clicking finish will import (1) the GKN classes themselves and (2) your generated plugin's classes.

Now, the packages of the GKN classes and your plugin show up in the left `Package Explorer` pane of Eclipse.


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Browse...
Select archive file:
Browse...
Projects:
Select All
Deselect All
Refresh
Options
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Cancel
Finish


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
/Users/dadi/workspace/development/GenericKnimeNodes
Browse...
Select archive file:
Browse...
Projects:
com.genericworkflownodes.knime.node_generator (/Users/dadi/workspace/develo
com.genericworkflownodes.knime.testing.feature (/Users/dadi/workspace/develop
com.genericworkflownodes.knime.tests (/Users/dadi/workspace/development/Ger
com.genericworkflownodes.knime.update (/Users/dadi/workspace/development/G
net.sf.samtools (/Users/dadi/workspace/development/GenericKnimeNodes/genera
net.sf.samtools.feature (/Users/dadi/workspace/development/GenericKnimeNodes
net.sf.samtools.linux.x86 (/Users/dadi/workspace/development/GenericKnimeNod
net.sf.samtools.linux.x86_64 (/Users/dadi/workspace/development/GenericKnimel
Select All
Deselect All
Refresh
Options
Search for nested projects
Copy projects into workspace
Hide projects that already exist in the workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Cancel
Finish


Package Explorer
Type Hierarchy
com.genericworkflownodes.knime
com.genericworkflownodes.knime.config
com.genericworkflownodes.knime.config.testing
com.genericworkflownodes.knime.feature
com.genericworkflownodes.knime.node_generator
com.genericworkflownodes.knime.testing.feature
com.genericworkflownodes.knime.tests
com.genericworkflownodes.knime.update
net.sf.samtools
net.sf.samtools.feature
net.sf.samtools.linux.x86
net.sf.samtools.linux.x86_64
net.sf.samtools.macosx.x86_64
net.sf.samtools.testing.feature
net.sf.samtools.win32.x86
net.sf.samtools.win32.x86_64

Tip: Synchronizing `ant` build result with Eclipse.

Since the code generation happens outside of Eclipse, there are often problems caused by Eclipse not recognizing updates in generated ‘’.java” files. After each call to `ant`, you should clean all built files in all projects by selecting the menu entries `Project > Clean...`, selecting `Clean all projects`, and then clicking `OK`.

Then, select all projects in the `Package Explorer`, right-click and select `Refresh`.

Tip: You might get a warning with in one of the KNIME files. In order to remove it you need to download the KNIME's test environment, but you can just ignore the error in our case.

Launching Eclipse with our Nodes

Finally, we have to launch KNIME with our plugin. We have to create a run configuration for this. Select `Run > Run Configurations...`.

In the `Run Configurations` window, select `Eclipse Application` on the left, then click the small `New launch configuration` icon on the top left (both marked in the following screenshot). Now, set the `Name` field to “KNIME”, select `Run an application` and select `org.knime.product.KNIME_APPLICATION` in the drop down menu. Finally, click `Run`.

Your tool will show up in the tool selector in `Community Nodes`.

Important: Sometimes KNIME complains about the Java version you are using. In that case, you can use Java 1.6. as shown in *Choosing The JRE Version*.

Important: If you are running a MacOS you might need to add *-Xms40m -Xmx512M -XX:MaxPermSize=256m -Xdock:icon=../Resources/Eclipse.icns -XstartOnFirstThread -Dorg.eclipse.swt.internal.carbon.smallFonts -server* to the `VM argument` box of your `Run Configuration`.

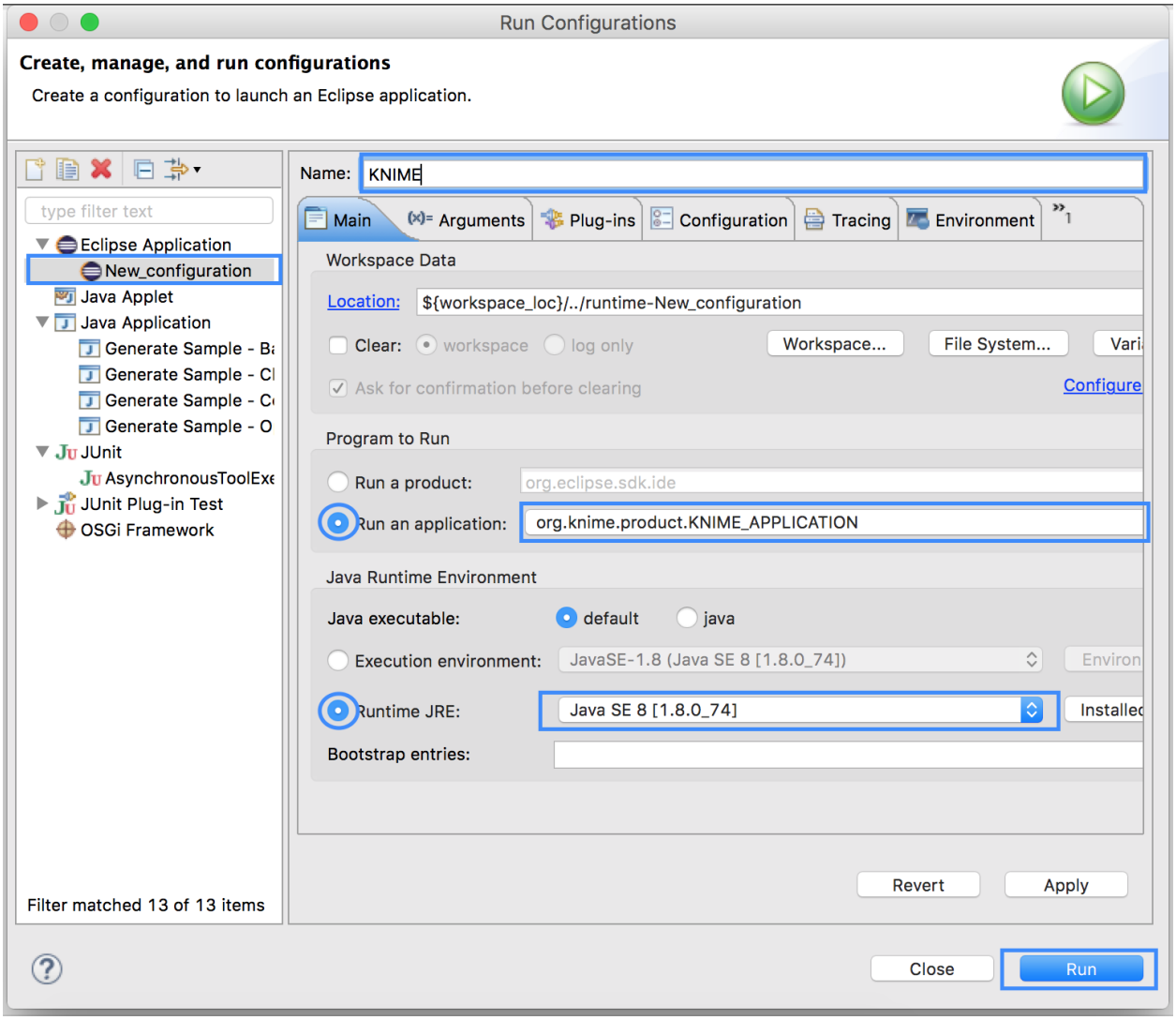
You should now be able to use the created node in a KNIME workflow.

If you would like to learn more about the structure of the plugin and where the crucial information is stored you can read the *Overview* section of the tutorial Generating KNIME Nodes.

Introduction

In bioinformatics Workflows contain an interconnected and orchestrated series of computational or data manipulation steps. In order to compose and execute such workflows one needs workflow management systems. Workflow management systems can represent how computation proceeds from one step to the next one in a form of directed graph in which nodes represent tasks to be executed and edges represent either the flow of data or dependencies between different tasks. Among the many workflow management systems suited for bioinformatics workflows we will consider two common frameworks that are used to create, manage and execute workflows. namely:

- KNIME and
- Galaxy


Run Configurations
Create, manage, and run configurations
Create a configuration to launch an Eclipse application.
type filter text
Eclipse Application
New_configuration
Java Applet
Java Application
Generate Sample - Ba
Generate Sample - Cl
Generate Sample - Cc
Generate Sample - O
JUnit
AsynchronousToolExe
JUnit Plug-in Test
OSGi Framework
Filter matched 13 of 13 items
Name: KNIME
Main
Arguments
Plug-ins
Configuration
Tracing
Environment
Workspace Data
Location: ${workspace_loc}/../runtime-New_configuration
Clear: workspace log only
Workspace...
File System...
Ask for confirmation before clearing
Configure
Program to Run
Run a product: org.eclipse.sdk.ide
Run an application: org.knime.product.KNIME_APPLICATION
Java Runtime Environment
Java executable: default java
Execution environment: JavaSE-1.8 (Java SE 8 [1.8.0_74])
Runtime JRE: Java SE 8 [1.8.0_74]
Bootstrap entries:
Revert
Apply
Close
Run

KNIME Workflows

In KNIME workflows are composed of (KNIME) nodes connected to each other by edges. The nodes are a representation of an application/algorithm that takes an input, process it and produces a desired output. The edges represent a flow of data from one specific application to the next one.

Generic KNIME nodes

KNIME nodes are usually shipped as eclipse plugins. The term Generic KNIME node refers to KNIME node (eclipse plugin) generated from any command line tool. This is done via **GenericKnimeNodes** (GWN) package which provides an infrastructure to automatically generate such nodes from the description of their command line.

Important:

- If you only want to use existing SeqAn apps in KNIME follow *Creating Workflows with SeqAn Nodes in KNIME*.
- If you want to learn how to convert any command-line-tool into a KNIME node read the tutorial *Generating KNIME Nodes*
- If you are a SeqAn application developer and you want to make your application KNIME ready follow the tutorial *Make Your SeqAn App KNIME Ready*
- If you want to learn how to generate a KNIME node out of a SeqAn application follow the tutorial *Generating SeqAn KNIME Nodes*

Galaxy Workflows

User Guide

Before you can start using the SeqAn library you have to make sure that it is installed and set up correctly. Optionally also install the dependencies:

- *Installing SeqAn*
- *Installing Dependencies*

Once this is done you have two choices for using SeqAn:

- *Using SeqAn in CMake-based projects*
- *Integration with your own Build System*

We highly recommend using the CMake-Module as it correctly handles many settings you otherwise need to set manually. It also provides the best cross-platform compatibility.

If you choose to use CMake, the following document contains more information on build configurations and using multiple build directories in parallel:

- *CMake Build Directories*

Some notes on using this manual

You will often see something like this in the manual:

```
# mkdir -p /tmp/mytempdir
```

This is called a terminal and you can find it on all major operating systems, sometimes also called "Konsole" or "Shell" (although that is something different strictly speaking).

You are expected to enter whatever comes right of the `#` (sometimes also the `$`) and then press `RETURN`. Knowing your way around the Terminal will make things easier, but it should be possible to just copy'n'paste.

Important: On Windows you are expected to be using the **PowerShell**, and not the legacy command prompt. Our tutorials will only work with the PowerShell!

ToC

Contents

Installing SeqAn

There are different ways to install SeqAn, we recommend to try these in the given order:

1. Native package management of the operating system.
2. Unpacking the library package from http://packages.seqan.de
3. Using the full sources from our github repository.

If possible, use the first option. If SeqAn is not available for your operating system, or if it is outdated, use the second option.

Use the third option if you want to use the master or develop branch which might contain bug-fixes and new features.

Native package management

SeqAn is available natively on the following platforms.

Tip: Before you install, please make sure that the version supplied is not completely out of date (a difference in the third digit is ok, but if the difference is bigger use the *Library Package* below). The current version of SeqAn is always shown on the SeqAn-Homepage and the version available on your platform is usually displayed in the info-link below.

Operating System		Package Name	Command	links
GNU/Linux	Arch			AUR
	Debian	seqan-dev	`apt install seqan-dev`	info | contact
	Fedora	seqan-devel	`yum install seqan-devel`	info | contact
	Gentoo	seqan	`emerge sci-biology/seqan`	info | contact
	Ubuntu	seqan-dev	`apt install seqan-dev`	info | contact
Mac	Homebrew	seqan	`brew install homebrew/science/seqan`	info | contact
	MacPorts	seqan	`port install seqan`	info | contact
BSD	FreeBSD	seqan	`pkg install seqan`	info | contact
	OpenBSD	seqan	`pkg_add seqan`	info | contact

You should execute the above commands in a terminal as the `root` user or prefix them with `sudo`. If you have problems installing the package on your operating system, or it is outdated, please write to the contact shown above (and replace `()` in the e-mail-address with `@`).

Library Package

First you need to download the most recent "library package" from http://packages.seqan.de and extract its contents. Now copy the *include* and *share* folders to their target location. This could be one of the following:

- `/usr/local` so they are available system-wide and automatically found by your program [requires root or sudo]
- `/opt/seqan` available system-wide and easy to remove again [requires root or sudo]
- `~/devel/seqan` some place in your home directory [does not require root or sudo]

In any case it is important to remember where you installed it to.

Full Sources

Make sure that you have git installed. For the operating systems mentioned above it can usually be achieved by using the respective command with *git* as package name.

For Windows there is Git client and shell available here.

Next create the required folders and clone our master branch:

```
~ # mkdir -p ~/devel
~ # cd ~/devel
~ # git clone https://github.com/seqan/seqan.git seqan
```

You can update this branch at a later point by running `git pull` in `~/devel/seqan`.

ToC

Contents

Installing Dependencies

SeqAn can optionally make use of ZLIB and BZip2. This is relevant mostly for Input/Output. Depending on your operating system you may need to install extra packages of these libraries or their headers.

GNU/Linux

It depends on your distribution whether these packages are installed by default or not.

On Debian, Ubuntu, Mint and similar distributions:

```
# sudo apt install zlib1g-dev libbz2-dev
```

Mac and BSD

Nothing needs to be done, the libraries and their headers are pre-installed.

Windows

The downloadable contribs contain precompiled library binaries (zlib, libbz2) for Windows by the supported compilers. The contribs come in 32 bit and 64 bit variants.

- Download contribs for 32 bit builds.
- Download contribs for 64 bit builds.

You can install both variants in parallel if you want to do both 32 bit and 64 bit builds. Previous contribs packages are available here and here you can find the code for building contribs with new VS versions, for example.

Now, extract the downloaded ZIP file either to `C:\Program Files` or `C:\`.

After downloading the 64 bit variant, you should now have a folder named `C:\Program Files\seqan-contrib-D20160115-x64` or a folder named `C:\seqan-contrib-D20130710-x64`.

After downloading the 32 bit variant, you should now have a folder named `C:\Program Files\seqan-contrib-D20160115-x86` or a folder named `C:\seqan-contrib-D20130710-x86`.

ToC

Contents

Using SeqAn in CMake-based projects

Overview

CMake is a cross-platform build system generator where you describe different executables, binaries and their dependencies in `CMakeLists.txt` files. Then, CMake generates build systems such as Makefiles or Visual Studio projects from these files. This article describes only the most basic things about CMake in general and focuses on how to use SeqAn easily from within CMake projects.

In CMake projects, one uses modules to find libraries such as SeqAn. SeqAn ships with such a module.

In the following we assume that you have installed CMake on your operating system. If you have not yet, install it via the operating systems mechanisms (see also *Setting up SeqAn*) and/or download from the CMake homepage.

You should also have a valid C++-Compiler installed. Refer to the GitHub-README to see which compilers are currently supported.

A Running Example

Create a folder somewhere, e.g. `~/devel/my_project` and in it the following two files:

`my_project.cpp`

```
#include <iostream>

#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

int main() {
    std::cout << CharString("Hello SeqAn!") << std::endl;
    return 0;
}
```

`CMakeLists.txt`

```
# Minimum cmake version
cmake_minimum_required (VERSION 3.0.0)

# Name of project and that it is C++ only.
project (my_project CXX)

# ----------------------------------------------------------------------------
# Dependencies
# ----------------------------------------------------------------------------

# Search for zlib as a dependency for SeqAn.
find_package (ZLIB)

# Load the SeqAn module and fail if not found.
find_package (SeqAn REQUIRED)

# ----------------------------------------------------------------------------
# Build Setup
# ----------------------------------------------------------------------------
```

```
# Add include directories.
include_directories (${SEQAN_INCLUDE_DIRS})

# Add definitions set by find_package (SeqAn).
add_definitions (${SEQAN_DEFINITIONS})

# Add CXX flags found by find_package (SeqAn).
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${SEQAN_CXX_FLAGS}")

# Add executable and link against SeqAn dependencies.
add_executable (my_project my_project.cpp)
target_link_libraries (my_project ${SEQAN_LIBRARIES})
```

Building The Project

First you should create a build directory, i.e. for cmake-builds everything happens in a different directory, than in the source directory. In our case create the directory `~/devel/my_project-build` and in there a folder `release`. More on why we use two levels *here*.

```
# mkdir -p ~/devel/my_project-build/release
# cd ~/devel/my_project-build/release
```

By default, the `cmake` program will look for `FindSeqAn.cmake` in its module directory. Usually, this is located in `/usr/share/cmake/Modules` or a similar location that is available system-wide. Depending on how you *installed SeqAn* it might be found by cmake automatically. If not, you have to give the path to cmake via the `CMAKE_MODULE_PATH` argument on the command line.

Also, CMake will look for the SeqAn include files in central locations such as `/usr/local/include`. Again, depending on your installation this might *just work*. If not, you need to specify the location via the `SEQAN_INCLUDE_PATH` argument.

When using operating system packages of SeqAn and the default compiler it might look like this:

```
# cmake ../../my_project
```

If you instead did a full git checkout to your home-directory in the previous step, it might look like this:

```
# cmake ../../my_project \
    -DCMAKE_MODULE_PATH=~/devel/seqan/util/cmake \
    -DSEQAN_INCLUDE_PATH=~/devel/seqan/include
```

Tip: Depending on your setup you might need to manually choose a more modern compiler! Please read *this page* for more information on configuring CMake builds. Don't forget to clean your CMake build directory after changing the compiler!

Finally you can then build the application by calling

- on Makefile-based builds (Linux/Mac/BSD):

  ```
  # make
  ```

- Windows

```
# cmake --build .
```

The above step is the only step you need to repeat when changing your source code. You only have to run CMake again, if you have changed the `CMakeLists.txt`.

You can then execute the application in the usual way

- on Makefile-based builds (Linux/Mac/BSD):

```
# ./my_project
```

- Windows

```
# my_project
```

Using IDEs

On Linux and BSD many IDEs directly support cmake, just open/import the `CMakeLists.txt` with e.g. KDevelop or QtCreator.

To use XCode on Mac with your CMake-based project, add `-G Xcode` to the cmake call above and then run `open TODO`.

On Windows a Visual Studio generator is used by default and you will find a `.vcxproj` in the source directory that you can open with Visual Studio.

See *this page* for more details.

Details of the FindSeqAn Module

As mentioned above, this line is the important line for including SeqAn:

```
find_package (SeqAn REQUIRED)
```

If SeqAn is only an optional dependency of your program, you can omit the `REQUIRED` keyword. In this case you should check the contents of the `SEQAN_FOUND` CMake-variable and depending on that configure your build, e.g. with custom Macros.

You can also check for the definition of SeqAn's version macros from within your code:

`SEQAN_VERSION_STRING` Concatenated version string, `${SEQAN_VERSION_MAJOR}.${SEQAN_VERSION_MINOR}.${SEQAN_VERSION_PATCH}`

`SEQAN_VERSION_MAJOR` Major version.

`SEQAN_VERSION_MINOR` Minor version.

`SEQAN_VERSION_PATCH` Patch-level version.

Dependencies

SeqAn itself has some optional dependencies. Certain features in SeqAn will be enabled or disabled, depending on whether the dependencies could be found.

Caution: Optional dependencies of SeqAn have to be searched **before** the SeqAn module is searched!

Currently, the following dependencies enable optional features:

ZLIB zlib compression library

BZip2 libbz2 compression library

OpenMP OpenMP language extensions to C/C++

An example of where you only want ZLIB and OpenMP support, but not BZip2, would look like this:

```
find_package (ZLIB)
find_package (OpenMP)
find_package (SeqAn)
```

From within CMake you can check the variables `ZLIB_FOUND` or `OpenMP_FOUND` to see the results of these dependency searches, but you can also use the following macros from within your source code to escape certain optional code paths:

SEQAN_HAS_ZLIB `TRUE` if zlib was found.

SEQAN_HAS_BZIP2 `TRUE` if libbz2 was found.

_OPENMP `TRUE` if OpenMP was found.

CMake build variables

As can be seen from the example above, the following variables need to be passed to `include_directories()`, `target_link_directories()`, and `add_definitions()` in your `CMakeLists.txt`:

SEQAN_INCLUDE_DIRS A list of include directories.

SEQAN_LIBRARIES A list of libraries to link against.

SEQAN_DEFINITIONS A list of definitions to be passed to the compiler.

Required additions to C++ compiler flags are in the following variable:

SEQAN_CXX_FLAGS C++ Compiler flags to add.

Caution: Please note that these variables include whatever has been added by the dependencies mentioned above so **do not add** e.g. `${OpenMP_CXX_FLAGS}` yourself!

ToC

Contents

Integration with your own Build System

This page gives an example of how to use SeqAn in your application based on your own Makefiles. You should be able to adapt the descriptions to configure your build system and/or IDE.

Tip: SeqAn is a header-only library.

Simply adding its include folder to your include path or installing it globally makes it available to your program.

C++14 Standard

On GNU/Linux, BSD and macOS, always add `-std=c++14` (or a newer standard) when building on the command line.

For XCode on macOS you need to set this option in the project settings.

As of Visual Studio 2015 our subset of C++14 is already available in all supported compilers.

OpenMP

On GNU/Linux, BSD and macOS, add `-fopenmp` unless you are using Clang versions older than 3.8.0.

For XCode on macOS OpenMP is not yet available.

With Visual Studio OpenMP is switched on by default.

Operating System specifics

GNU/Linux

Libraries

Add `-lrt -lpthread` to the compiler call.

BSD

Libraries

Add `-lpthread -lexecinfo -lelf` to the compiler call.

Misc

Also define `-D_GLIBCXX_USE_C99=1` if you are using gcc-4.9.

Warning levels

It is recommended to compile your programs with as many warnings enabled as possible. This section explains which flags to set for different compilers.

GCC, Clang, ICC (unix)

The following flags are recommended:

`-W -Wall -pedantic` Maximal sensitivity of compiler against possible problems.

Visual Studio

For Visual Studio, the following flags are recommended:

```
/W2 /wd4996 -D_CRT_SECURE_NO_WARNINGS
```

Explanation:

`/W2` Warning level 2 is pretty verbose already. In the future, we will support level 3 without warnings in SeqAn code.

`/wd4996` Allows the use of some deprecated functions without warnings.

`-D_CRT_SECURE_NO_WARNINGS ::“` Some C functions like `sprintf` are prone to incorrect usage and security holes. Replacing such calls does not have a high priority right now since SeqAn is usually not used on servers facing the outside world.

Preprocessor Defines Affecting SeqAn

There are certain preprocessor symbols that affect the behaviour of SeqAn.

SEQAN_ENABLE_DEBUG

possible value 0, 1

default 0

meaning If set to 1, assertions within SeqAn (`SEQAN_ASSERT...`) are enabled, they are disabled otherwise. Is forced to 1 if `SEQAN_ENABLE_TESTING` is true. This flag will internally always correspond to the inverse of `NDEBUG`, i.e. setting it to 1 will force `NDEBUG` to be undefined and setting it to 0 will forcefully set `NDEBUG`.

SEQAN_ENABLE_TESTING

possible value 0, 1

default 0

meaning This makes the code very slow, and should only be used when running the SeqAn unit tests. Has to be set to 1 for tests to work.

SEQAN_ASYNC_IO

possible value 0, 1

default 0 on `FreeBSD/32Bit` and `OpenBSD/*`; 1 otherwise

meaning Whether asynchronous input/output is available.

SEQAN_HAS_EXECINFO

possible value 0, 1

default depends on platform / existance of `<execinfo.h>`

meaning This should almost always be set to 1 on non-Windows platforms!

SEQAN_HAS_BZIP2

possible value 0, 1

default 0

meaning If set to 1 then libbzip2 is expected to be available. You have to link against the library (e.g. add `-lbz2` to your linke rflags) and `bzlib.h` must be in your include path.

SEQAN_HAS_ZLIB

possible value 0, 1

default 0

meaning If set to 1 then zlib is expected to be available. You have to link against the library (e.g. add `-lz` to your linker flags) and `zlib.h` must be in your include path.

Settings Projects Using Seqan

You normally want to have at least two build modes: one for debugging and one for optimized compiling. The following settings have to be applied to your IDE project/Makefiles (below is an example for a Makefile based project).

Debug Builds

Besides enabling debug symbols and disabling optimization, there are the following SeqAn specific settings to be applied.

- Add SeqAn to your include path
- Define `SEQAN_ENABLE_DEBUG` to be `1`.

This translates into the following GCC flags:

```
-g -O0 -DSEQAN_ENABLE_DEBUG=1 -I${PATH_TO_SEQAN_INSTALL}/include
```

Release/Optimized Builds

Besides disabling debug symbols, enabling optimization and disabling assertions in the standard library, there are the following SeqAn specific settings to be applied.

- Add SeqAn to your include path
- Define `NDEBUG`. This will make sure that `SEQAN_ENABLE_DEBUG` is 0 and also other STL includes of your program are not slowed down.

This translates into the following GCC flags:

```
-O3 -DNDEBUG -I${PATH_TO_SEQAN_INSTALL}/include
```

Caution: While some guides tell you to not use `-O3` this is absolutely crucial for SeqAn based applications to perform well. Unoptimized builds are slower by multiple factors!

An Example Project Based On Makefiles

We will create a project with good old Makefiles and GCC. The program will not do much but can serve as a minimal example on how to use SeqAn with your own build process. You should be able to adapt this guide to your favourite build system or IDE.

The example project can be found in `util/makefile_project`. The project layout looks like this:

```
.
|-- Makefile.rules
|-- Makefile
|-- README
|-- debug
|   `-- Makefile
|-- release
|   `-- Makefile
`-- src
    `-- main.cpp
```

main.cpp

We have one directory `src` for source files. The file `main.cpp` looks as follows:

```
#include <seqan/basic.h>
#include <seqan/sequence.h>
#include <seqan/stream.h>

using namespace seqan;

int main()
{
    std::cout << CharString("Hello SeqAn!") << std::endl;
    return 0;
}
```

It includes SeqAn headers just as you would within the SeqAn CMake framework.

Now, consider the contents of the Makefiles:

Makefile.rules

Contains the necessary commands to build the object file for the program `main.cpp` and then make an executeable `main` from it and clean targets. This file is included from the files `release/Makefile` and `debug/Makefile`.

```
SRC=../src
CXXFLAGS+=-I../include -std=c++14

default: all
all: main

main: main.o
        $(CXX) $(LDFLAGS) -o main main.o

main.o: $(SRC)/main.cpp
        $(CXX) $(CXXFLAGS) -c -o main.o $(SRC)/main.cpp

clean:
        rm -f main.o main

.PHONY: default all clean
```

Makefile

Allows to build both debug and release builds by calling `make debug`, `make release` or `make all` from the project directory. Removes all binaries with `make clean`.

```
default: all

all: debug release

debug:
        $(MAKE) -C debug

release:
```

```
        $(MAKE) -C release

clean:
        $(MAKE) -C debug clean
        $(MAKE) -C release clean

.PHONY: default all debug release clean
```

debug/Makefile, release/Makefile

The file `debug/Makefile` looks as follows.

```
include ../Makefile.rules

CXXFLAGS+=-g -O0 -DSEQAN_ENABLE_DEBUG=1
```

The file `release/Makefile` looks as follows.

```
include ../Makefile.rules

CXXFLAGS+=-O3 -DNDEBUG
```

These Makefiles include the file `Makefile.rules`. They add build type specific arguments to the variables `$(CXXFLAGS)`. For debug builds, debug symbols are enabled, optimization level 0 is chosen, testing is enabled in SeqAn and debugging is disabled. For release builds, debug symbols are not, optimization level 3 is chosen, testing and debugging are both disabled in SeqAn. For good measure, we also disable assertions in the C library with `-DNDEBUG`.

Notes

Above we added the include path to SeqAn's include directory manually. By changing the include path, we can install the SeqAn library anywhere. For example, we could create a directory `include` parallel to `src`, copy the release version of SeqAn into it and then change the include path of the compiler to point to this directory (value `../include`).

Short Version

OS	Compiler	Flags
Linux	GCC/Clang3.8/ICC	`-I /path/to/seqan/include -std=c++14 -O3 -DNDEBUG -W -Wall -pedantic -fopenmp -lpthread -lrt`
BSD	GCC/Clang3.8/ICC	`-I /path/to/seqan/include -std=c++14 -O3 -DNDEBUG -W -Wall -pedantic -fopenmp -lpthread -lexecinfo -lelf -D_GLIBCXX_USE_C99=1`
macOS	system's Clang	`-I /path/to/seqan/include -std=c++14 -O3 -DNDEBUG -W -Wall -pedantic`
Windows	Visual Studio MSVC	`/W2 /wd4996 -D_CRT_SECURE_NO_WARNINGS`

Adapt the include path to the actual place of SeqAn's `include` folder!

ToC

Contents

CMake Build directories in detail

Motivation

Why would you need more than one build directory or more than one IDE project file? This is very useful

- if you want to use the same set of source files from multiple version of the same IDE (e.g. two Visual Studio versions),
- if you want to have both debug builds (for debugging) and release builds (for performance tests) in parallel,
- if you have your source files stored on a shared network location and want to have build files on two computer and/or operating systems, or
- if you want to build the sources with two different compilers or compiler versions at the same time (e.g. to see whether you can figure out compiler errors better from the messages by another compiler).

The overall idea is very simple: you create one build directory for each variant and call CMake in each of it using different settings.

Tip: A nice side-effect of separating source and build directories is also that you can just delete you build directory and recreate it if you feel that something went wrong configuring your build.

CMake Parameters

A central question with CMake is the choice of the so called generator. Enter `cmake -G` to get a list of the supported ones. The most common generators are the **Unix Makefiles** which are default on Linux/Mac/BSD. But there are also specific generators for IDEs, such as Visual Studio, XCode or CodeBlocks.

For most of the IDEs further choices like "Release or Debug" are available from the graphical user interface of the IDE, whereas, for the Unix Makefile generator, we can specify the *build types* using a command line option. Also, the compiler program (and version) can be switched using a command line option.

Examples

We assume that your project source is at `~/devel/my_project`.

Unix Makefiles

Different compilers:

```
# mkdir -p ~/devel/my_project-build/release_gcc5
# cd ~/devel/my_project-build/release_gcc5
# cmake ../../my_project -DCMAKE_CXX_COMPILER=g++-5
[...]
# mkdir -p ~/devel/my_project-build/release_clang37
# cd ~/devel/my_project-build/release_clang37
# cmake ../../my_project -DCMAKE_CXX_COMPILER=clang++-3.7
[...]
```

Please note that the above only works if your compiler is in your PATH. You can instead also specify a full path like `-DCMAKE_CXX_COMPILER=/opt/local/bin/g++-mp-4.9`.

Debug and release builds:

```
# mkdir -p ~/devel/my_project-build/release
# cd ~/devel/my_project-build/release
# cmake ../../my_project
[...]
# mkdir -p ~/devel/my_project-build/debug
# cd ~/devel/my_project-build/debug
# cmake ../../my_project -DCMAKE_BUILD_TYPE=Debug
[...]
```

Of course the above can also be combined to have `debug_clang37` et cetera.

Visual Studio

Different versions (please note that versions older than 2015 are not supported any longer):

```
# mkdir -p ~/devel/my_project-build/vs2015
# cd ~/devel/my_project-build/vs2015
# cmake ../../my_project -G "Visual Studio 14 2015"
[...]
# mkdir -p ~/devel/my_project-build/vs2013
# cd ~/devel/my_project-build/vs2013
# cmake ../../my_project -G "Visual Studio 12 2013"
[...]
```

32Bit and 64Bit:

```
# mkdir -p ~/devel/my_project-build/vs2015_32
# cd ~/devel/my_project-build/vs2015_32
# cmake ../../my_project -G "Visual Studio 14 2015"
[...]
# mkdir -p ~/devel/my_project-build/vs2015_64
# cd ~/devel/my_project-build/vs2015_64
# cmake ../../my_project -G "Visual Studio 14 2015 Win64"
[...]
```

Caution: 64Bit builds on Windows

You almost always want 64Bit builds when using SeqAn, so don't forget to specify a generator that ends in "Win64". It is not the default, even on 64Bit Windows installations.

Different Compilers:

Intel Compiler 2016:

```
# mkdir -p ~/devel/my_project-build/intel_32
# cd ~/devel/my_project-build/intel_32
# cmake ../../my_project -G "Visual Studio 14 2015" -T "Intel C++ Compiler 16.0"
[...]

# mkdir -p ~/devel/my_project-build/intel_64
# cd ~/devel/my_project-build/intel_64
# cmake ../../my_project -G "Visual Studio 14 2015 Win64" -T "Intel C++ Compiler 16.0"
[...]
```

Clang/C2 3.7 or 3.8 (requires CMake 3.6):

```
# mkdir -p ~/devel/my_project-build/clang_c2_32
# cd ~/devel/my_project-build/clang_c2_32
# cmake ../../my_project -G "Visual Studio 14 2015" -T "v140_clang_3_7"
[...]

# mkdir -p ~/devel/my_project-build/clang_c2_64
# cd ~/devel/my_project-build/clang_c2_64
# cmake ../../my_project -G "Visual Studio 14 2015 Win64" -T "v140_clang_3_7"
[...]
```

Note: If Clang/C2 3.8 is installed, the tool-chain name in Visual Studio 14 is still "v140_clang_3_7" even though the name says otherwise.

Caution: Clang/C2 is currently experimental and shouldn't be used in production.

XCode

```
# mkdir -p ~/devel/my_project-build/xcode
# cd ~/devel/my_project-build/xcode
# cmake ../../my_project -G "Xcode"
[...]
```

Contributer Guide

SeqAn is on GitHub: http://github.com/seqan/seqan

Use the GitHub page to fork SeqAn, create tickets and/or pull requests. You can also follow and like the project there!

Contributing Code or Documentation

If you are unfamiliar with git, you need to learn about it first. See the Atlassian Git Tutoial for an introduction to Git.

Next learn about the specific Git Workflow that we use and how we mark commits:

- *Git Workflow*
- *Writing Commit Messages*

If you are just changing something small, try to follow the style of whatever you are changing. If you contribute more code, please take the time to read:

- *C++ Code Style*
- *Other Code Styles*

SeqAn's documentation system, called **dox**, is similar to doxygen, but not identical. Read about it here if you want to contribute documentation (all code should be documented!):

- *API Documentation System (dox)*

ToC

Contents

Git Workflow

Getting Started

Install the command line client, download a GUI and have a look at the basic Atlassian tutorial.

GUI

- SourceTree - Windows or MacOS X.
- Gitg - Linux/GNOME.

Documentation

- Atlassian git tutorials - easy and recommended.
- Official git manual - complete but more advanced.

Clone the SeqAn repository

SeqAn is hosted on GitHub. Execute the following command to get the last sources:

```
~ # git clone https://github.com/seqan/seqan.git seqan-src
```

SeqAn Workflow

The SeqAn workflow is based on the Gitflow workflow by Atlassian. The workflow is based on two persistent branches: master and develop. Development of new library and app features usually occurs on develop. The master branch receives only new library and app releases, in addition to hot-fixes to previous releases. Thus, the master branch is always stable and safe to use, and the develop branch contains the last development but might occasionally break overnight. The most frequent development use cases are documented below.

Develop a feature in a module or an app

Follow the steps in "Mary and John begin new features" and "Mary finishes her feature".

- Create a new feature branch based on develop.
- Perform your changes and commit them onto your feature branch.
- When the development is complete, push the feature branch to your repository on GithHub.
- Create a GitHub pull request to develop.
- Delete your feature branch once it has been merged.

Fix an existing bug in a module or app

Follow the steps in "End-user discovers a bug".

- Create a new hotfix branch based on master.
- Perform your changes and commit them onto your hotfix branch.
- **When the fix is ready, push your hotfix branch to repository on GitHub. Then:**
    1. Create a GitHub pull request to master.
    2. Create a GitHub pull request to develop.
    3. The pull requests should contain only the commits from your hotfix branch.
- Delete your hotfix branch once it has been merged through the pull request.

Develop new modules and apps

Create a new module or app branch where to develop your new module or application. The branch should be based on master if your module or application doesn't rely on any recently developed features. If a new feature becomes necessary later on, the branch can be rebased onto develop. When the development is complete, the branch can be merged back into the corresponding base branch - either master or develop.

Rules

- Never push feature branches to the SeqAn repository.
- Submit code reviews through GitHub.

ToC

Contents

Writing Commit Messages

Format

On every commit to our revision control system (currently SVN) please provide a commit message of the following form:

```
[CLASS1,CLASS2,...] Short description

Optional long description
```

- The first line starts with an arbitrary number of tags in square brackets, e.g. `[CLASS1]` or `[CLASS1, CLASS2]`. See below for a possible list of classes.
- These tags are followed by a short description, try to keep the first line below 120 characters, 80 if possible.
- You can add an optional long description after an empty line.

Possible Classes

NOP
: Only whitespace changes.
: e.g. removed trailing whitespace, replaced tabs by spaces, changed indentation

DOC
: Changes in the user documentation.
: This includes changes to the DDDoc documentation, README files etc.

COMMENT
: Changes in the source documentation.
: These changes are not visible to the users.
: This includes `TODO(${name}):` statements.

API
: Changes in the API.
: These changes classically break backward compatibility.
: e.g. renaming of function names, changing of function parameter order.

INTERNAL
: Changes in the implementation.
: These changes do not influence the public the API.
: e.g. renaming of variable names, simplification of code

FEATURE
: A user-visible feature.
: e.g. extension of an interface, measurable performance improvement
: *If the change is also API breaking the classes FEATURE* **and** *API must be used.*

FIX
: Bug removal.
: If one or more bugs from the ticket tracker are removed then this should be written as `[FIXED-#7,#35]` where `#7` and `#35` are ticket numbers.

TEST
: Addition or changes of tests.
: All code changes that are accompanied with tests must provide the original and the TEST class.
: Don't consider this as a coercion but as a privilege to use both classes!

CLI
: Change to the command line interface of a program.
: e.g. change to the arguments a program accepts, change to the messages printed to the user
: *Output that is meant for debugging or detailed introspection is handled by the LOG class.*

LOG
: Change of output for developers or very advanced users.
: This is the output that is meant for debugging or detailed introspection that is excluded from `CLI`.
: Such output is usually printed to `stderr`.

Examples

Example: API Changes

API change with tests and detailed description of changes.

```
[API,TEST] Large changes of align module's API.

This is a large patch that mostly updates the align module:

* The Anchor Gaps specialization is moved from the store module to the align module.
* The Array Gaps class is rewritten.
* Both Anchor and Array gaps have mostly the same API now, differences are documented.
* Greatly unified the interface of the ``globalAlignment()`` and ``localAlignment()``␣
↪interface.
* The ``LocalAlignmentFinder`` is now called ``LocalAlignmentEnumerator``.
* Threw out unused DP algorithms (DP algorithm will be cleaned up in the future, see␣
↪below).
* Clipping of gaps works consistently in both Anchor and Array Gaps data structures.
* Wrote a large number of tests for all changes.
* Adjusted SeqAn library and apps to new API.
```

```
All in all, this should make the module more usable, albeit breaking the interface in␣
↪some cases.
There will be a future change by Rene that strongly unifies the DP algorithms.
This will not inflict another API change, though, and further simplify the align␣
↪module.
```

Example: Bug Fixes

A fix that solves two tickets:

```
[FIX-#240,#356] Fixed iteration of ``ModifiedString``s.

Quite involved fix that allows iteration of ``ModifiedString`` objects.
```

A fix that does not have a ticket:

```
[FIX] Fixed reading of CIGAR string in module bam_io.

There was a bug when reading the operation "F", which was translated to
FLABBERGASTED.  Fixed this to the documented behavior.
```

Example: Internal Changes

An internal change, reordering of code without changing the public API.

```
[INTERNAL] Reordering code in module sequence so no more generated forwards are␣
↪needed.
```

An internal change might include tests and improved comments.

```
[INTERNAL,TEST,COMMENTS] Greatly improved transmogrify module.

Restructured the whole internal structure of the module, adding a large number of␣
↪tests
and improving the source-level documentation.  The user level documentation is still
lacking and should be the target of a future change.
```

Example: Changes To Command Line Interface And Logging

Changes to the command line interface:

```
[CLI] Changed output of STELLAR such to unify scientific notation floats.
```

Changes to logging in an app:

```
[LOG] Improved logging in RazerS 5.

Much more detailed logging allows easier debugging.  Part of this should probably be
commented out before the next stable release once the dust has settled and most
bugs have been removed.
```

ToC

Contents

C++ Code Style

The aim of this style guide is to enforce a certain level of canonicity on all SeqAn code. Besides good comments, having a common style guide is the key to being able to understand and change code written by others easily.

(The style guide partially follows the Google C++ Code Style Guide.)

C++ Features

Reference Arguments

We prefer reference arguments to pointer arguments. Use `const` where possible.

Use C-Style Logical Operators

Use `&&`, `||`, and `!` instead of `and`, `or`, and `not`.

While available from C++98, MSVC does not support them out of the box, a special header `<iso646.h>` has to be included. Also, they are unfamiliar to most C++ programmers and nothing in SeqAn is using them.

Default Arguments

Default arguments to global functions are problematic with generated forwards. They can be replaced with function overloading. So do not use them!

You can replace default arguments with function overloading as follows. Do not do this.

```
inline double f(int x, double y = 1.0)
{
    // ...
}
```

Do this instead.

```
inline double f(int x, double y)
{
    // ...
}

inline double f(int x)
{
    return f(x, 1.0);
}
```

Exceptions

SeqAn functions throw exceptions only to report unrecoverable errors, usually during I/O. Instead, functions expected to either success or fail use boolean return values to report their status.

Virtual Member Functions

SeqAn heavily uses template subclassing instead of C++ built-in subclassing. This technique requires using global member functions instead of in-class member functions.

If the design requires using in-class member functions, the keyword `virtual` should be avoided. Virtual member functions cannot be inlined and are thus slow when used in tight loops.

static_cast<>

Prefer `static_cast<>` to C-style casts.

const_cast<>

Use const-casts only to make an object const. Do not remove consts. Rather, use the `mutable` keyword on selected members. `const_cast<>` is allowed for interfacing with external (C) APIs where the `const` keyword is missing but which do not modify the variable.

The following is an example where `const_cast<>` is OK:

```
template <typename T>
bool isXyz(T const & x)
{
    return x._member == 0;
}

template <typename T>
bool isXyz(T & x)
{
    return const_cast<T const &>(x)._member == 0;
}
```

reinterpret_cast<>

Only use `reinterpret_cast<>` when you absolutely have to and you know what you are doing! Sometimes, it is useful for very low-level code but mostly it indicates a design flaw.

pre/post increment/decrement

Prefer the "pre" variants for decrement and increment, especially in loops. Their advantage is that no copy of an object has to be made.

Good:

```
typedef Iterator<TContainer>::Type TIterator;
for (TIterator it = begin(container); atEnd(it); ++it)
{
    // do work
}
```

Bad:

```
typedef Iterator<TContainer>::Type TIterator;
for (TIterator it = begin(container); atEnd(it); it++)
{
    // do work
}
```

Code Quality

Const-Correctness

Write const correct code. Read the C++ FAQ const correctness article for more information. Besides other things, this allows to use temporary objects without copying in functions that do not need to change their arguments.

Compiler Warnings

All code in the repository must compile without any warnings using the flags generated by the CMake system.

Currently, the GCC flags are:

:: -W -Wall -Wstrict-aliasing -pedantic -Wno-long-long -Wno-variadic-macros

Style Conformance

Follow this code style whenever possible. However, prefer consistency to conformance.

If you are editing code that is non-conforming consider whether you could/should adapt the whole file to the new style. If this is not feasible, prefer consistency to conformance.

Semantics

Parameter Ordering

The general parameter order should be (1) output, (2) non-const input (e.g. file handles), (3) input, (4) tags. Within these groups, the order should be from mandatory to optional.

In SeqAn, we read functions `f(out1, out2, out3, ..., in1, in2, in3, ...)` as `(out1, out2, out3, ...) <- f(in1, in2, in3, ...)`.

E.g. `assign()`:

```
template <typename T>
void f(T & out, T const & in)
{
    out = in;
}
```

Scoping, Helper Code

Global Variables

Do not use global variables. They introduce hard-to find bugs and require the introduction of a link-time library.

Tags In Function Arguments

Tags in function arguments should always be const.

```
// somewhere in your code:

struct Move_;
typedef Tag<Move_> Move;

// then, later:

void appendValue(TContainer, Move const &)
{
    // ...
}
```

Structs and Classes

Visibility Specifiers

Visibility specifiers should go on the same indentation level as the `class` keyword.

Example:

```
class MyStruct
{
public:
protected:
private:
};
```

Tag Definitions

Tags that are possibly also used in other modules must not have additional parameters and be defined using the `Tag<>` template. Tags that have parameters must only be used within the module they are defined in and have non-generic names.

Tags defined with the `Tag<>` template and a typedef can be defined multiply. These definitions must have the following pattern:

```
struct TagName_;
typedef Tag<TagName_> TagName;
```

This way, there can be multiple definitions of the same tag since the struct `TagName_` is only declared but not defined and there can be duplicate typedefs.

For tags (also those used for specialization) that have template parameters, the case is different. Here, we cannot wrap them inside the `Tag<>` template with a typedef since it still depends on parameters. Also we want to be able to instantiate tags so we can pass them as function arguments. Thus, we have to add a struct body and thus define the

struct. There cannot be multiple identical definitions in C++. Thus, each tag with parameters must have a unique name throughout SeqAn. Possibly too generic names should be avoided. E.g. `Chained` should be reserved as the name for a global tag but `ChainedFile<>` can be used as a specialization tag in a file-related module.

Note that this restriction does not apply for internally used tags (e.g. those that have an underscore postfix) since these can be renamed without breaking the public API.

In-Place Member Functions

Whenever possible, functions should be declared and defined outside the class. The constructor, destructor and few operators have to be defined inside the class, however.

The following has to be defined and declared within the class (also see Wikipedia):

- constructors
- destructors
- function call operator `operator()`
- type cast operator `operator T()`
- array subscript operator `operator[]()`
- dereference-and-access-member operator `operator->()`
- assignment operator `operator=()`

Formatting

Constructor Initialization Lists

If the whole function prototype fits in one line, keep it in one line. Otherwise, wrap line after column and put each argument on its own line indented by one level. Align the initialization list.

Example:

```
class Class
{
    MyClass() :
        member1(0),
        member2(1),
        member3(3)
    {}
};
```

Line Length

The maximum line length is 120. Use a line length of 80 for header comments and the code section separators.

Non-ASCII Characters

All files should be UTF-8, non-ASCII characters should not occur in them nevertheless.

In comments, use `ss` instead of ß and `ae` instead of ä etc.

In strings, use UTF-8 coding. For example, `"\xEF\xBB\xBF"` is the Unicode zero-width no-break space character, which would be invisible if included in the source as straight UTF-8.

Spaces VS Tabs

Do not use tabs! Use spaces. Use `"\t"` in strings instead of plain tabs.

After some discussion, we settled on this. All programmer's editors can be configured to use spaces instead of tabs. We use four spaces instead of a tab.

There can be problems when indenting in for loops with tabs, for example. Consider the following (`-->|` is a tab, _ is a space):

```
for (int i = 0, j = 0, k = 0, ...;
_____cond1 && cond2 &&; ++i)
{
  // ...
}
```

Here, indentation can happen up to match the previous line. Mixing tabs and spaces works, too. However, since tabs are not shown in the editor, people might indent a file with mixed tabs and spaces with spaces if they are free to mix tabs and spaces.

```
for (int i = 0, j = 0, k = 0, ...;
-->|_cond1 && cond2 &&; ++i)
{
  // ...
}
```

Indentation

We use an indentation of four spaces per level.

Note that ''namespaces do not cause an increase in indentation level.''

```
namespace seqan {

class SomeClass
{
};

}  // namespace seqan
```

Trailing Whitespace

Trailing whitespace is forbidden.

Trailing whitespace is not visible, leading whitespace for indentation is perceptible through the text following it. Anything that cannot be seen can lead to "trash changes" in the GitHub repository when somebody accidentally removes it.

Inline Comments

Use inline comments to document variables.

Possibly align inline comments.

```
short x;     // a short is enough!
int myVar;   // this is my variable, do not touch it
```

Brace Positions

Always put brace positions on the next line.

```
class MyClass
{
public:
    int x;

    MyClass() : x(10)
    {}
};

void foo(char c)
{
    switch (c)
    {
    case 'X':
        break;
    }
    // ...
}
```

Conditionals

Use no spaces inside the parantheses, the `else` keyword belongs on a new line, use block braces consistently.

Conditional statements should look like this:

```
if (a == b)
{
    return 0;
}
else if (c == d)
{
    int x = a + b + d;
    return x;
}

if (a == b)
    return 0;
else if (c == d)
    return a + b + d;
```

Do not leave out the spaces before and after the parantheses, do not put leading or trailing space in the paranthesis. The following is wrong:

```
if (foo){
    return 0;
}
if(foo)
    return 0;
if (foo )
    return 0;
```

Make sure to add braces to all blocks if any block has one. The following is wrong:

```
if (a == b)
    return 0;
else if (c == d)
{
    int x = a + b + d;
    return x;
}
```

Loops and Switch Statements

Switch statements may use braces for blocks. Empty loop bodies should use `{}` or `continue`.

Format your switch statements as follows. The usage of blocks is optional. Blocks can be useful for declaring variables inside the switch statement.

```
switch (var)
{
case 0:
    return 1;
case 1:
    return 0;
default:
    SEQAN_FAIL("Invalid value!");
}

switch (var2)
{
case 0:
    return 1;
case 1:
{
    int x = 0;
    for (int i = 0; i < var3; ++i)
        x ++ i;
    return x;
}
default:
    SEQAN_FAIL("Invalid value!");
}
```

Empty loop bodies should use `{}` or `continue`, but not a single semicolon.

```
while (condition)
{
  // Repeat test until it returns false.
}
```

```
for (int i = 0; i < kSomeNumber; ++i)
    {}  // Good - empty body.
while (condition)
    continue;  // Good - continue indicates no logic.
```

Expressions

Binary expressions are surrounded by one space. Unary expressions are preceded by one space.

Example:

```
if (a == b || c == d || e == f || !x)
{
    // ...
}
bool y = !x;
unsigned i = ~j;
```

Type Expressions

No spaces around period or arrow. Add spaces before and after pointer and references. `const` comes after the type.

The following are good examples:

```
int x = 0;
int * ptr = x;                      // OK, spaces are good.
int const & ref = x;                // OK, const after int
int main(int argc, char ** argv);   // OK, group pointers.
```

Bad Examples:

```
int x = 0;
int* ptr = x;          // bad spaces
int *ptr = x;          // bad spaces
const int & ref = x;   // wrong placement of const
int x = ptr -> z;      // bad spaces
int x = obj. z;        // bad spaces
```

Function Return Types

If a function definition is short, everything is on the same line. Otherwise, split.

Good example:

```
int foo();

template <typename TString>
typename Value<TString>::Type
anotherFunction(TString const & foo, TString const & bar, /*...*/)
{
    // ...
}
```

Inline Functions

If a function definition is short, everything is on the same line. Otherwise put inline and return type in the same line.

Good example:

```
inline int foo();

template <typename TString>
inline typename Value<TString>::Type
anotherFunction(TString const & foo, TString const & bar, /*...*/)
{
    // ...
}
```

Function Argument Lists

If it fits in one line, keep in one line. Otherwise, wrap at the paranthesis, put each argument on its own line. For very long function names and parameter lines, break after opening bracket.

Good example:

```
template <typename TA, typename TB>
inline void foo(TA & a, TB & b);

template </*...*/>
inline void foo2(TA & a,
                 TB & b,
                 ...
                 TY & y,
                 TZ & z);

template </*...*/>
inline void _functionThisIsAVeryVeryLongFunctionNameSinceItsAHelper(
    TThisTypeWasMadeToForceYouToWrapInTheLongNameMode & a,
    TB & b,
    TC & c,
    TB & d,
    ...);
```

Template Argument Lists

Follow conventions of function parameter lists, no blank after opening <.

As for function parameters, try to fit everything on one line if possible, otherwise, break the template parameters over multiple lines and put the commas directly after the type names.

```
template <typename T1, typename T1>
void foo() {}

template <typename T1, typename T2, ...
          typename T10, typename T11>
void bar() {}
```

Multiple closing > go to the same line and are only separated by spaces if two closing angular brackets come after each other.

```
typedef Iterator<Value<TValue>::Type,
                 Standard> ::Type

typedef String<char, Alloc<> > TMyString
// -------------------------^
```

Function Calls

Similar rules as in *Function Argument Lists* apply. When wrapped, not each parameter has to occur on its own line.

Example:

```
foo(a, b);

foo2(a, b, c, ...
     x, y, z);

if (x)
{
    if (y)
    {
        _functionThisIsAVeryVeryLongFunctionNameSinceItsAHelper(
            firstParameterWithALongName, b, c, d);
    }
}
```

Naming Rules

In the following, camel case means that the first letter of each word is written upper case, the remainder is written in lower case. Abbreviations of length 2 are kept in upper case, longer abbreviations are camel-cased.

Macros

Macros are all upper case, separated by underscores, prefixed with `SEQAN_`.

Example:

```
SEQAN_ASSERT_EQ(val1, val2);

#define SEQAN_MY_TMP_MACRO(x) f(x)
// ...
SEQAN_MY_TMP_MACRO(1);
// ...
#undef SEQAN_MY_TMP_MACRO
```

Variable Naming

Variables are named in camel case, starting with a lower-case parameter. Internal member variables have an underscore prefix.

Example:

```
int x;
int myVar;
int saValue(/*...*/);
int getSAValue(/*...*/);

struct FooBar
{
    int _x;
};
```

Constant / Enum Value Naming

Constant and enum values are named like macros: all-upper case, separated by dashes.

Example:

```
enum MyEnum
{
    MY_ENUM_VALUE1 = 1,
    MY_ENUM_VALUE2 = 20
};

int const MY_VAR = 10;
```

Struct / Enum / Class Naming

Types are written in camel case, starting with an upper case character.

Internal library types have an underscore suffix.

Example:

```
struct InternalType_
{};

struct SAValue
{};

struct LcpTable
{};
```

Metafunction Naming

Metafunctions are named like structs, defined values are named `VALUE`, types `Type`.

Metafunctions should not export any other types or values publically, e.g. they should have an underscore suffix.

Example:

```
template <typename T>
struct MyMetaFunction
{
    typedef typename RemoveConst<T>::Type TNoConst_;
```

```
    typedef TNonConst_ Type;
};

template <typename T>
struct MyMetaFunction2
{
    typedef True Type;
    static bool const VALUE = false;
};
```

Function Naming

The same naming rule as for variables applies.

Example:

```
void fooBar();

template <typename T>
int saValue(T & x);

template <typename T>
void lcpTable(T & x);
```

Names In Documentation

In the documentation, classes have the same name as in the source code, e.g. the class StringSet is documented as "class StringSet." Specializations are named "`$SPEC $CLASS`", e.g. "Concat StringSet", "Horspool Finder."

Comments

File Comments

Each file should begin with a file header.

The file header has the format. The `skel.py` tool automatically generates files with appropriate headers.

```
// ==========================================================================
//                                $PROJECT NAME
// ==========================================================================
// Copyright (C) 2010 $AUTHOR, $ORGANIZATION
//
// $LICENSE
//
// ==========================================================================
// Author: $NAME <$EMAIL>
// ==========================================================================
// $FILE_DESCRIPTION
// ==========================================================================
```

Class, Function, Metafunction, Enum, Macro DDDoc Comments

Each public class, function, metafunction, enum, and macro should be documented using *dox API docs*. Internal code should be documented, too.

Example:

```
/*!
 * @class IntervalAndCargo
 * @headerfile <seqan/refinement.h>
 * @brief A simple record type that stores an interval and a cargo value.
 *
 * @signature template <typename TValue, typename TCargo>
 *            struct IntervalAndCargo;
 *
 * @tparam TValue The value type of the record, defaults to int.
 * @tparam TCargo The cargo type of the record, defaults to int.
 */

template <typename TValue = int, typename TCargo = int>
class IntervalAndCargo
{
    // ...
};

// This functions helps the XYZ class to fulfill the ABC functionality.
//
// It corresponds to function FUNC() in the paper describing the original
// algorithm.  The variables in this function correspond to the names in the
// paper and thus the code style is broken locally.

void _helperFunction(/*...*/)
{}
```

Implementation Comments

All functions etc. should be well-documented. In most cases, it is more important how something is done instead of of what is done.

TODO Comments

TODO comments have the format `// TODO($USERNAME): $TODO_COMMENT`. The username is the username of the one writing the item, not the one to fix it. Use GitHub issues for this.

Source Tree Structure

File Name Rules

File and directories are named all-lower case, words are separated by underscores.

Exceptions are `INFO`, `COPYING`, `README`, ... files.

Examples:

- string_base.h
- string_packed.h
- suffix_array.h
- lcp_table.h

File Structure

Header #define guard

The header #define include guards are constructed from full paths to the repository root.

Example:

filename	preprocessor symbol
seqan/include/seqan/basic/iterator_base.h	SEQAN_INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_

```
#ifndef SEQAN_INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_
#define SEQAN_INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_
#endif  // #ifndef SEQAN_INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_
```

Include Order

The include order should be (1) standard library requirements, (2) external requirements, (3) required SeqAn modules.

In SeqAn module headers (e.g. *basic.h*), then all files in the module are included.

CPP File Structure

```
// ==========================================================================
//                                 $APP_NAME
// ==========================================================================
// Copyright (c) 2006-2016, Knut Reinert, FU Berlin
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
//     * Redistributions of source code must retain the above copyright
//       notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above copyright
//       notice, this list of conditions and the following disclaimer in the
//       documentation and/or other materials provided with the distribution.
//     * Neither the name of Knut Reinert or the FU Berlin nor the names of
//       its contributors may be used to endorse or promote products derived
//       from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL KNUT REINERT OR THE FU BERLIN BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
```

```
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
// LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
// OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
// DAMAGE.
//
// ============================================================================
// Author: $AUTHOR_NAME <$AUTHOR_EMAIL>
// ============================================================================
// $FILE_COMMENT
// ============================================================================

#include <seqan/basic.h>
#include <seqan/sequence.h>

#include "app_name.h"

using namespace seqan;

// Program entry point
int main(int argc, char const ** argv)
{
    // ...
}
```

Application Header Structure

```
// ============================================================================
//                                $APP_NAME
// ============================================================================
// Copyright (c) 2006-2016, Knut Reinert, FU Berlin
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
//     * Redistributions of source code must retain the above copyright
//       notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above copyright
//       notice, this list of conditions and the following disclaimer in the
//       documentation and/or other materials provided with the distribution.
//     * Neither the name of Knut Reinert or the FU Berlin nor the names of
//       its contributors may be used to endorse or promote products derived
//       from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL KNUT REINERT OR THE FU BERLIN BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
// LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
// OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
// DAMAGE.
```

```
//
// ==========================================================================
// Author: $AUTHOR_NAME <$AUTHOR_EMAIL>
// ==========================================================================
// $FILE_COMMENT
// ==========================================================================

#ifndef APPS_APP_NAME_HEADER_FILE_H_
#define APPS_APP_NAME_HEADER_FILE_H_

// ==========================================================================
// Forwards
// ==========================================================================

// ==========================================================================
// Tags, Classes, Enums
// ==========================================================================

// ----------------------------------------------------------------------------
// Class ClassName
// ----------------------------------------------------------------------------

// ==========================================================================
// Metafunctions
// ==========================================================================

// ----------------------------------------------------------------------------
// Metafunction MetafunctionName
// ----------------------------------------------------------------------------

// ==========================================================================
// Functions
// ==========================================================================

// ----------------------------------------------------------------------------
// Function functionName()
// ----------------------------------------------------------------------------

#endif  // APPS_APP_NAME_HEADER_FILE_H_
```

Library Header Structure

```
// ==========================================================================
//                 SeqAn - The Library for Sequence Analysis
// ==========================================================================
// Copyright (c) 2006-2016, Knut Reinert, FU Berlin
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
//     * Redistributions of source code must retain the above copyright
//       notice, this list of conditions and the following disclaimer.
//     * Redistributions in binary form must reproduce the above copyright
//       notice, this list of conditions and the following disclaimer in the
//       documentation and/or other materials provided with the distribution.
```

```
//     * Neither the name of Knut Reinert or the FU Berlin nor the names of
//       its contributors may be used to endorse or promote products derived
//       from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL KNUT REINERT OR THE FU BERLIN BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
// LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
// OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
// DAMAGE.
//
// ==========================================================================
// Author: AUTHOR NAME <AUTHOR EMAIL>
// ==========================================================================
// SHORT COMMENT ON WHAT THIS FILE CONTAINS
// ==========================================================================

#ifndef INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_
#define INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_

namespace seqan {

// ==========================================================================
// Forwards
// ==========================================================================

// ==========================================================================
// Tags, Classes, Enums
// ==========================================================================

// ----------------------------------------------------------------------------
// Class ClassName
// ----------------------------------------------------------------------------

// ==========================================================================
// Metafunctions
// ==========================================================================

// ----------------------------------------------------------------------------
// Metafunction MetafunctionName
// ----------------------------------------------------------------------------

// ==========================================================================
// Functions
// ==========================================================================

// ----------------------------------------------------------------------------
// Function functionName()
// ----------------------------------------------------------------------------

}  // namespace seqan

#endif  // INCLUDE_SEQAN_BASIC_ITERATOR_BASE_H_
```

ToC

Contents

Other Style Guides

Python Style Guide

Some very few points:

- Follow PEP 8.
- Use single-quotes for strings, i.e. `'this is a string'` and double-quotes for docstrings, e.g. `"""This is a docstring."""`.
- Name functions, classes, constants as in SeqAn, variables and member variables are named `lower_case_with_underscores`.

JavaScript Style Guide

Follow the *SeqAn C++ Style Guide* in spirit.

ToC

Contents

API Documentation System (dox)

Since the 1.4.1 release, SeqAn uses a new documentation system. The syntax is similar to Doxygen but slightly different. The main differences are (1) not identifying functions by their signatures but only by their names, (2) adding the idea of metafunctions, (3) adding the idea of interface functions and (4) an extension to SeqAn-specific things like documenting concepts.

General Documentation Structure

Dox comments are placed in C-style comments with an exclamation mark (see below). The first dox tag should be placed on the next line, each line should begin with a correctly indented star. The first line only contains the slash-star-exclamation-mark and the last line only contains the star-slash.

```
/*!
 * @fn myFunction
 * @signature void myFunction()
 */
```

The documentation and the code are independent. Each item to be documented (adaption, class, concept, enum, function, group, macro, metafunction, page, tag, typedef, variable) has to be epxlicitely given (see tags below). The available top level tags are [#adaption @adaption], [#class @class], [#concept @concept], [#defgroup @defgroup], [#enum @enum], [#fn @fn], [#macro @macro], [#metafunction @mfn], [#page @page], [#tag @tag], [#typedef @typedef], and [#variable @var].

Each top-level tag creates a documentation entry. For example, the following defines a class `Klass` with two global interface functions `f1` and `f2` for this class:

```
/*!
 * @class Klass
 * @fn Klass#f1
 * @fn Klass#f2
 */
```

Member functions are given using `::`, the same as in the C++ language:

```
/*!
 * @class Klass
 * @fn Klass::memberFunc
 */
```

Global interface functions are global functions that belong to the interface of a type. Similar, interface metafunctions are metafunctions that belong to the interface of a type. Their fully qualified name for dox consists of the type name, followed by a hash `#` and the function/metafunction name:

```
/*!
 * @class Klass
 * @fn Klass#interfaceFunc
 * @mfn Klass#InterfaceMetaFunc
 */
```

Below the top-level tags, come the second-level tags. The first kind of second-level tags defines properties of an entry. Important such second-level entries are `@brief`, `@signature`, `@see`, `@param`, `@tparam`, `@return`. You can also write text for the description of your entity and use tags such as `@section`, `@subsection`, `@snippet`, `@code` to format the description. You can use HTML tags for formatting the documentation.

Example:

```
/*!
 * @class Align
 * @brief Store a tabular alignment.
 *
 * @signature template <typename TSource, typename TSpec>
 *            class Align;
 *
 * @tparam TSource The type of the underlying sequence.
 * @tparam TSpec   Tag for selecting the specialization of the Align class.
 *
 * The <tt>Align</tt> class provides a tabular alignment of sequences with the
 * same type.  The sequences are given with <tt>TSource</tt>.  An <tt>Align</tt>
 * object will use a <a href="seqan:Gaps">Gaps</a> object for each sequence.
 * The specialization of the <a href="seqan:Gaps">Gaps</a> object can be selected
 * using the <tt>TSpec</tt> template parameter.
 *
 * @see Gaps
 * @see globalAlignment
 */
```

Images are included using `<img src="${PATH}">` where `${PATH}` is relative to the source image directory.

Tag Documentation

Below, we differentiate between **names** and **labels**.

Names are used to identify documentation items and must follow extended C++ identifier rules. A sub name consists of only alphanumeric characters and the underscore is allowed, must not start with a number. Sub names can be glued together with `::` for class members and `#` for interface functions. In contrast, **labels** are used for the display to the user. For example, the alloc string has the name `AllocString` but the label "Alloc String", the constructor of `AllocString` has name `AllocString::String`, and its length function has name `AllocString#length`.

@adaption

Signature `@adaption AdaptionName [Adaption Label]`

Top-level tag.

Defines an adaption with the given name and an optional label.

An adaption is a collection of global interface functions and metafunctions that adapt a type outside the SeqAn library to a concept in the SeqAn library. For example, the STL `std::string` class can be adapted to the interface of the `StringConcept` concept.

```
/*!
 * @adaption StdStringToStringConcept std::string to Sequence concept
 * @brief The <tt>std::string</tt> class is adapted to the Sequence concept.
 */
```

@aka

Signature `@aka OtherName`

Second-level entry.

Assigns an alias name for a function, metafunction, class, concept, or enum. The list of aliases will be printed for each code entry. Also, the aliases will be incorporated into search results.

```
/*!
 * @class InfixSegment
 * @brief Represents a part of a string.
 *
 * @aka substring
 */

template <typename TSequence>
class InfixSegment<TSequence, Infix>;
```

@brief

Signature `@brief Brief description.`

Second-level tag.

Defines the brief description of the top-level entry it belongs to. You can use HTML in the description.

```
/*!
 * @fn f
 * @brief A minimal function.
 * @signature void f();
 */

void f();
```

@class

Signature `@class ClassName [Class Label]`

Top-level tag.

Defines a class with the given name `ClassName` and an optional label.

```
/*!
 * @class AllocString Alloc String
 * @extends String
 * @brief Implementation of the String class using dynamically allocated array.
 *
 * @signature template <typename TAlphabet, typename TSpec>
 * class String<TAlphabet, Alloc<TSpec> >;
 * @tparam TAlphabet Type of the alphabet (the string's value).
 * @tparam TSpec     Tag for the further specialization.
 */

template <typename TAlphabet, typename TSpec>
class String<TAlphabet, Alloc<TSpec> >
{
    // ...
};
```

@code

Signature `@code{.ext} ... @endcode`

Second-level tag.

Provides the means to display code blocks in the documentation. The extension `.ext` is used for identifying the type (use `.cpp` for C++ code) and selecting the appropriate highlighting.

```
/*!
 * @fn f
 * @brief Minimal function.
 * @signature void f();
 *
 * @code{.cpp}
 * int main()
 * {
 *     f();  // Call function.
 *     return 0;
 * }
 * @endcode
 */

void f();
```

Note that you can use the extension value `.console` to see console output.

```
/*!
 * @fn f
 * @brief Some function
 *
 * @section Examples
 *
 * @include demos/module/demo_f.cpp
 *
 * The output is as follows:
 *
 * @code{.console}
 * This is some output of the program.
 * @endcode
  */
```

@concept

Signature `@concept ConceptName [Concept Label]`

Top-level tag.

Creates a documentation entry for a concept with the given name and an optional label. All concept names should have the suffix `Concept`. Use the fake keyword `concept` in the `@signature`.

A concept is the C++ equivalent to interfaces known in other classes. C++ provides no real way for concepts so at the moment they are a formal construct used in the documentation.

```
/*!
 * @concept StringConcept Sequence
 * @signature concept StringConcept;
 * @extends ContainerConcept
```

```
 * @brief Concept for sequence types.
 */
```

@defgroup

Signature `@defgroup GroupName [Group Label]`

Top-level tag.

Creates a documentation entry for a group with a given name and an optional label. Groups are for rough grouping of global functions and/or tags.

You can put types and functions into a group similar to making global interface functions and metafunctions part of the interface of a class or concept.

```
/*!
 * @defgroup FastxIO FASTA/FASTQ I/O
 * @brief Functionality for FASTA and FASTQ I/O.
 *
 * @fn FastxIO#readRecord
 * @brief Read one record from FASTA/FASTQ files.
 *
 * @fn FastxIO#writeRecord
 * @brief Write one record to FASTA/FASTQ files.
 *
 * @fn FastxIO#readBatch
 * @brief Read multiple records from FASTA/FASTQ file, limit to a given count.
 *
 * @fn FastxIO#writeBatch
 * @brief Write multiple records to FASTA/FASTQ file, limit to a given count.
 *
 * @fn FastxIO#readAll
 * @brief Read all records from a FASTA/FASTQ file.
 *
 * @fn FastxIO#writeAll
 * @brief Write all records to a FASTA/FASTQ file.
 */
```

@deprecated

Signature `@deprecated message`

Second-level entry.

Marks a given function, metafunction, class, concept, or enum as deprecated. A deprecation message will be generated in the API documentation.

```
/*!
 * @fn f
 * @deprecated Use @link g @endlink instead.
 * @brief Minimal function.
 */

void f();
```

@enum

Signature `@enum EnumName [Enum Label]`

Top-level entry.

Provides documentation for an enum with given name and optional label.

```
/*!
 * @enum MyEnum
 * @brief An enum.
 *
 * @val MyEnum VALUE1
 * @brief VALUE1 value of enum MyEnum.
 *
 * @val MyEnum VALUE2
 * @brief VALUE2 value of enum MyEnum.
 */

enum MyEnum
{
  VALUE1,
  VALUE2
};
```

@extends

Signature `@extends OtherName`

Gives a parent class for a given class or a parent concept for a given concept.

```
/*!
 * @concept OneConcept
 *
 * @concept TwoConcept
 * @extends OneConept
 *
 * @class MyClass
 *
 * @class OtherClass
 * @extends MyClass
 */
```

@fn

Signature `@fn FunctionName [Function Label]`

Top-level entry.

Documents a function (global, global interface, or member) with given name and label. The type of the function is given by its name.

```
/*!
 * @fn globalAlignment
 * @brief Pairwise, DP-based global alignment.
 */
```

@headerfile

Signature `@headerfile path`

Second-level entry.

Gives the required `#include` path for a code entry.

Note: Use angular brackets as below for SeqAn includes.

```
/*!
 * @fn f
 * @brief A minimal function.
 * @headerfile <seqan/module.h>
 */
```

@implements

Signature `@implements ConceptName`

Second-level entry.

Marks a class to implement a given concept.

```
/*!
 * @concept MyConcept
 *
 * @class ClassName
 * @implements MyConcept
 */
```

@include

Signature `@include path/to/file`

Second-level entry.

Includes a C++ source file as an example. See [#snippet @snippet] for including fragments.

```
/*!
 * @fn f
 * @brief Minimal function.
 *
 * The following example shows the usage of the function.
 * @include demos/use_f.cpp
 */
```

@internal

Signature `@internal [ignored comment`

Second-level entry.

Marks a given function, metafunction, class, concept, or enum as internal. You can also provide a comment that is ignored/not used in the output.

```
/*!
 * @fn f
 * @internal
 * @brief Minimal function.
 */

void f();
```

@link

Signature `@link TargetName target label`

In-text tag.

Provides tag to link to a documentation entry with a given label.

The difference to [#see @see] is that `@link .. @endlink` is used inline in text whereas `@see` is a second-level tag and adds a `see` property to the documented top-level entry. Use `@link` to link to entries within the documentation and the HTML `<a>` tag to link to external resources.

```
/*!
 * @fn f
 * @brief Minimal function.
 *
 * The function is mostly useful with the @link String string class@endlink.
 */
```

@macro

Signature `@macro MacroName [Macro Label]`

Top-level tag.

Documents a macro.

```
/*!
 * @macro MY_MACRO
 * @brief Multiply two values.
 *
 * @signature #define MY_MACRO(i, j) ...
 * @param i A value for i.
 * @param j A value for j.
 * @return The product of i and j: (i * j)
 */

#define MY_MACRO(i, j) (i * j)
```

@mfn

Signature `@mfn MetafunctionName [Metafunction Label]`

Top-level tag.

Documents a metafunction.

```
/*!
 * @mfn Identity
 * @brief Identity function for types.
 *
 * @signature Identity<T>::Type
 * @tparam T The type to pass in.
 * @returns The type T.
 */

template <typename T>
struct Identity
{
    typedef T Type;
};
```

@note

Signature `@note message`

Second-level entry.

Adds an informative note to a function, metafunction, class, concept, enum, or group.

```
/*!
 * @fn f
 * @note Very useful if used together with @link g @endlink.
 * @brief Minimal function.
 */

void f();
```

@page

Signature `@page PageName [Page Title]`

Top-level entry.

Creates a documentation page.

```
/*!
 * @page SomePage Page Title
 *
 * A very simple page
 *
 * @section Section
 *
 * A section!
 *
 * @subsection Subsection
 *
 * A subsection!
 */
```

@param

Signature `@param Name Label`

Second-level entry.

Documents a value (and non-type) parameter from a function or member function.

```
/*!
 * @fn square
 * @brief Compute the square of an <tt>int</tt> value.
 *
 * @signature int square(x);
 * @param x The value to compute square of (type <tt>int</tt>).
 * @return int The square of <tt>x</tt>.
 */

int square(int x);
```

@return

Signature `@return Type Label`

Defines the return value for a function or metafunction.

Also see the example for [#param @param].

When documenting functions and the result type is the result of a metafunction then use a `TXyz` return type in `@return` and document `TXyz` in the text of `@return` as follows:

```
/*!
 * @fn lengthSquare
 * @brief Compute the square of the length of a container.
 *
 * @signature TSize square(c);
 *
 * @param c The container to compute the squared length of.
 * @return TSize squared length of <tt>c</tt>.  <tt>TSize</tt> is the size type of
↪<tt>c</tt>.
 */

template <typename TContainer>
typename Size<TContainer>::Type lengthSquare(TContainer const & c);
```

@throw

Signature `@return Exception Label`

Adds a note on a function or macro throwing an exception.

```
/*!
 * @fn myFunction
 * @brief Writes things to a file.
 * @signature void myFunction(char const * filename);
 *
 * @param[in] filename File to write to.
```

```
 *
 * @throw std::runtime_error If something goes wrong.
 */
void myFunction(char const * filename);
```

@datarace

Signature `@datarace Description`

Describes possible data races for functions and macros. If this value is not specified it defaults to `Thread safety unknown!`

```
/*!
 * @fn myFunction
 * @brief Writes things to a file.
 * @signature void myFunction(char const * filename);
 *
 * @param[in] filename File to write to.
 *
 * @datarace This function is not thread safe and concurrent writes to the file might␣
↪invalidate the output.
 */
void myFunction(char const * filename);
```

@section

Signature `@section Title`

Second-level entry.

Adds a section to the documentation of an entry.

See the example for [#page @page].

@see

Signature `@see EntryName`

Second-level entry.

Adds “see also” link to a documentation entry.

```
/*!
 * @fn f
 * @brief A simple function.
 *
 * Here is a snippet:
 *
 * @snippet demos/use_f.cpp Simple Function
 */
```

And here is the file with the snippet.

```
/* Some code */

int main(int argc, char const ** argv)
{
//![Simple Function]
    return 0;
//![Simple Function]
}

/* Some more code */
```

@tag

Signature `@tag TagName`

Top-level entry.

Documents a tag. Mostly, you would group tags in a group using [#defgroup @defgroup].

```
/*!
 * @defgroup MyTagGroup My Tag Group
 *
 * @tag MyTagGroup#TagName
 * @tag MyTagGroup#MyOtherTagName
 */
```

@tparam

Signature `@tparam TArg`

Second-level entry.

Documents a template parameter of a metafunction or class template.

```
/*!
 * @mfn MetaFunc
 * @signature MetaFunc<T1, T2>::Type
 *
 * @tparam T1 First type.
 * @tparam T2 Second type.
 */
```

@typedef

Signature `@typedef TypedefName`

Top-level entry.

Documents a typedef.

```
/*!
 * @typedef CharString
 * @brief An AllocString of character.
 *
```

```
 * @signature typedef String<char, Alloc<> > CharString;
 */
```

@var

Signature `@var VariableType VariableName`

Top-level entry. Document a global variable or member variable.

```
/*!
 * @class MyClass
 *
 * @var int MyClass::iVar
 */

class MyClass
{
public:
    int iVar;
};
```

@val

Signature `@val EnumType EnumValueName`

Top-level entry. Documents an enum value.

```
/*!
 * @enum EnumName
 * @brief My enum.
 * @signature enum EnumName;
 *
 * @val EnumName::VALUE1;
 * @brief The first enum value.
 *
 * @val EnumName::VALUE2;
 * @brief The second enum value.
 */

enum MyEnum
{
    VALUE1,
    VALUE2
};
```

@warning

Signature `@warning message`

Second-level entry.

Adds a warning to a function, metafunction, class, concept, enum, or group.

```
/*!
 * @fn f
 * @note Using this function can lead to memory leaks.  Try to use @link g @endlink␣
↪instead.
 * @brief Minimal function.
 */

void f();
```

Best Practice

This section describes the best practice when writing documentation.

Clarifying Links

Our usability research indicates that some functionality is confusing (e.g. see #1050) but cannot be removed. One example is the function `reserve()` which can be used to *increase* the *capacity* of a container whereas the function `resize()` allows to change the *size* of a container, *increasing or decreasing* its size.

The documentation of such functions should contain a clarifying text and a link to the other function.

```
/*!
 * @fn Sequence#reserve
 *
 * Can be used to increase the <b>capacity</b> of a sequence.
 *
 * Note that you can only modify the capacity of the sequence.  If you want to modify␣
↪the
 * <b>length</b> of the sequence then you have to use @link Sequence#resize @endlink.
 */
```

Documentation Location

Add the documentation where it belongs. For example, when documenting a class with multiple member functions, put the dox comments for the class before the class, the documentation of the member functions in front of the member functions. For another example, if you have to define multiple signatures for a global interface function or metafunctions, put the documentation before the first function.

```
/*!
 * @class Klass
 * @brief A class.
 */
class Klass
{
public:
    /*!
     * @var int Klass::x
     * @brief The internal value.
     */
    int x;

    /*!
     * @fn Klass::Klass
```

```
     * @brief The constructor.
     *
     * @signature Klass::Klass()
     * @signature Klass::Klass(i)
     * @param i The initial value for the member <tt>x</tt> (type <tt>int</tt>).
     */
    Klass() : x(0)
    {}

    Klass(int x) : x(0)
    {}

    /*!
     * @fn Klass::f
     * @brief Increment member <tt>x</tt>
     * @signature void Klass::f()
     */
    void f()
    {
        ++x;
    }
};
```

Signatures

Always document the return type of a function. If it is the result of a metafunction or otherwise depends on the input type, use `TResult` or so and document it with `@return`.

HTML Subset

You can use inline HTML to format your description and also for creating links.

- Links into the documentation can be generated using `<a>` if the scheme in `href` is `seqan::` `<a href="seqan:AllocString">the alloc string</a>`.
- Use `<i>` for italic/emphasized text.
- Use `<b>` for bold text.
- Use `<tt>` for typewriter text.

Tag Ordering

Keep consistent ordering of second-level tags. The following order should be used, i.e. if several of the following tags appear, they should appear in the order below.

1. `@internal`
2. `@deprecated`
3. `@warning`
4. `@note`
5. `@brief`
6. `@extends`

7. `@implements`
8. `@signature`
9. `@param`
10. `@tparam`
11. `@return`
12. `@headerfile`
13. The documentation body with the following tags in any order (as fit for the documentation text) and possibly interleaved with text: `@code`, `@snippet`, `@include`, `@section`, `@subsection`.
14. `@see`
15. `@aka`

Documenting Concepts

All concepts should have the suffix `Concept`.

Use the pseudo keyword `concept` in the `@signature`.

Use the following template:

```
/*!
 * @concept MyConcept
 * @brief The concept title.
 *
 * @signature concept MyConcept;
 *
 * The concept description possibly using include, snippet, and <b><i>formatting</i></
↪b> etc.
 */
```

Documenting Classes

Use the following template:

```
/*!
 * @class AllocString Alloc String
 * @brief A string storing its elements on dynamically heap-allocated arrays.
 *
 * @signature template <typename TAlphabet, typename TSpec>
 * class AllocString<TAlphabet, Alloc<TSpec> >;
 * @tparam TAlphabet The alphabet/value type to use.
 * @tparam TSpec     The tag to use for further specialization.
 *
 * The class description possibly using include, snippet, and <b><i>formatting</i></b>
↪ etc.
 */
```

Documenting Functions

Use the following template:

```
/*!
 * @fn globalAlignment
 * @brief Global DP-based pairwise alignment.
 *
 * @signature TScore globalAlignment(align, scoringScheme);
 * @signature TScore globalAlignment(align, scoringScheme, lowerBand, upperBand);
 * @param align Align object to store the result in. Must have length 2 and be filled␣
↪with sequences.
 * @param scoringScheme Score object to use for scoring.
 * @param lowerBand The lower band of the alignment (<tt>int</tt>).
 * @param upperBAnd The upper band of the alignment (<tt>int</tt>).
 * @return TScore The alignment score of type <tt>Value<TScore>::Type</tt> where <tt>
↪TScore</tt> is the type of <tt>scoringScheme</tt>.
 *
 * The function description possibly using include, snippet, and <b><i>formatting</i>
↪</b> etc.
 */
```

Documenting Metafunctions

Use the following template:

```
/*!
 * @mfn Size
 * @brief Return size type of another type.
 *
 * @signature Size<T>::Type
 * @tparam T The type to query for its size type.
 * @return TSize The size type to use for T.
 *
 * The class description possibly using include, snippet, and <b><i>formatting</i></b>
↪ etc.
 */
```

Documenting Enums

```
/*!
 * @enum EnumName
 * @brief My enum.
 * @signature enum EnumName;
 *
 * @var EnumName::VALUE
 * @summary The enum's first value.
 *
 * @var EnumName::VALUE2
 * @summary The enum's second value.
 */
```

Difference to Doxygen

If you already know Doxygen, the following major differences apply.

- The documentation is more independent of the actual code. Doxygen creates a documentation entry for all functions that are present in the code and allows the additional documentation, e.g. using `@fn` for adding functions. With the SeqAn dox system, you have to explicitely use a top level tag for adding documentation items.
- Documentation entries are not identified by their signature but by their name.
- We allow the definition of interface functions and metafunctions (e.g. `@fn Klass#func` and `@mfn Klass#Func`) in addition to member functions (`@fn Klass::func`).
- We do not allow tags with backslashes but consistently use at signs (`@`).

Team Guide

This guide is aimed at all people who contribute to SeqAn on a regular basis, and especially those that are responsible for certain parts of it. Downstream package maintainers should also read some of the articles, especially the *Releases page*.

It includes a description of the repository structure, detailed conventions and instructions, as well as documentation of the nightly build system.

ToC

Contents

The SeqAn Repository

This article describes the SeqAn repository structure and how to work with full SeqAn sources.

Getting Started

We assume that you have read *Installing SeqAn* and have cloned or unzipped the **full SeqAn sources** to `~/devel/seqan` (not the "library sources" described in other places).

SeqAn supports the usual CMake build types and we recommend *using multiple build directories*. Start like this:

```
# mkdir -p ~/devel/seqan-build/release
# cd ~/devel/seqan-build/release
# cmake ../../seqan -DCMAKE_BUILD_TYPE=Release
```

In addition to `CMAKE_BUILD_TYPE` there is also the `SEQAN_BUILD_SYSTEM` parameter which can be one of

1. `DEVELOP` – all build targets (apps, demos, tests) and documentation (dox, manual) are created (the default).
2. `SEQAN_RELEASE_LIBRARY` – only dox and library targets are created.
3. `SEQAN_RELEASE_APPS` – all app targets are created, but nothing else.
4. `APP:$APPNAME` – only a single app target is created for the chosen app.

All build systems other than `DEVELOP` are only relevant to *packaging releases*.

As usual, calling `make $TARGET` will build a single target and just `make` will build all targets. On Windows, run `cmake --build . --target $TARGET` or just `cmake --build` instead.

Overview

The main repository structure is shown in the following picture.

```
seqan
  |-- CMakeLists.txt    CMake script file.
  |
  |-- LICENSE           Top-Level Information Files
  |-- README.rst
  |
  |-- apps              Applications
  |
  |-- demos             Demos
  |
  |-- dox               API documentation system
  |
  |-- include/seqan     SeqAn header ("the library")
  |
  |-- manual            Manuals
  |
  |-- tests             Unit tests for library modules
  |
  `-- util              Miscellaneous and Utility Code
```

The repository root contains some **information files** such as the `LICENSE` and `README.rst`. The other folders are as follows:

apps

The apps folder contains many applications, and each application directory contains one `CMakeLists.txt` file and the files for compiling at least one binary. Usually, apps have tests, too. In this case, there is a subdirectory `tests`. Writing application tests is covered in detail in the article *Writing App Tests*.

The general structure of an app is as follows:

```
seqan/apps/razers
  |-- CMakeLists.txt      CMake script file
  |
  |-- README              Documentation and License Files
  |-- LICENSE
  |
  |-- example             Small Example Files
  |     |-- genome.fa
  |     |-- reads.fa
  |     `-- ...
```

```
  |
  |-- razers.cpp          Source Files for Executables
  |-- razers.h
  |-- ...
  |
  `-- tests               App Tests Files
```

Note that some applications have binary names (make targets) that are not identical to the app-name, e.g. yara has `yara_mapper` and `yara_indexer`.

demos

The demos are short programs and code snippets that are used in the dox or the manual. They serve as small examples and also functions as additional unit tests.

dox

The SeqAn API documentation is created using a customly-written system called *dox*. It is very similar to doxygen, you can find out more about the syntax in *API Documentation System (dox)*.

You can build the documentation in the *dox* subfolder of the *source folder*:

```
~   # cd ~/devel/seqan/dox
dox # ./dox_only.sh
```

This will build the documentation into the sub directory `html`.

include/seqan

This is the actual library consisting of multiple modules:

```
include/
  |-- seqan/
  |     |-- basic/                      Library Module basic
  |     |     |-- aggregate_concept.h
  |     |     |-- debug_test_system.h
  |     |     `-- ...
  |     |-- basic.h
  |     |
  |     |-- sequence/                   Library Module sequence
  |     |-- sequence.h
  |     |
  |     `-- ...                         Other Library Modules
```

On the top level, there is the folder `seqan` that contains the library modules. Inside the folder `seqan`, there is one directory and one header for each module.

The folder `<module-name>` contains the headers for the module module-name. The header `<module-name>.h` includes the headers from the module module-name. Including the header makes the code in the module available.

Note: Header only library

Remember that SeqAn is a template library that consists entirely of headers. No build steps are required for building the library and no shared objects will be created.

---

manual

The SeqAn manual is created using the Sphinx documentation system.

Follow these instructions to set up a local sphinx environment and build the manual:

```
$ virtualenv ~/seqan-manual-env
$ source ~/seqan-manual-env/bin/activate
(seqan-manual-env) $ cd ~/seqan/manual
(seqan-manual-env) $ pip install -r requirements.txt
(seqan-manual-env) $ make html
```

Note that you have to first build the dox documentation since plugins for generating the `:dox:` links rely on the generated search index for checks. In order to get correct dox-links within the generated manuals, you have to specify the correct branch version. If you are working on the develop branch there is nothing to do, since `'develop'` is set by default. But if you are working on another branch, for example `master`, you can set the correct branch by calling

```
(seqan-manual-env) $ export READTHEDOCS_VERSION='master'
```

before you call `make html` as described in the previous step. This will generate the correct links to the master's version of the dox, i.e., `http://docs.seqan.de/seqan/master/`

tests

The folder `tests` contains the unit tests for the library modules.

For each library module, there is a directory below `tests` with the same name that contains the tests for this module. Simpler modules have one tests executable, whereas there might be multiple tests executables for larger modules. For example, the module `index` has multiple test programs `test_index_qgram`, `test_index_shapes` etc. Writing tests is explained in detail in the article *Writing Unit Tests*.

ToC

Contents

Writing Unit Tests

This page describes how to write tests for the SeqAn library. Each test program defines a *Test Suite*, a collection of related *Tests*.

Test Suite Skeleton / Example

A skeleton and example for a test suite program looks as follows:

```
#include <seqan/basic.h>

SEQAN_DEFINE_TEST(test_suite_name_test_name)
{
    int ii = 1;
    for (int jj = 0; jj < 10; ++jj)
    {
        ii *= 2;
    }
    SEQAN_ASSERT_EQ(ii, 1024);
}

SEQAN_BEGIN_TESTSUITE(test_suite_name)
{
    SEQAN_CALL_TEST(test_suite_name_test_name);
}
SEQAN_END_TESTSUITE
```

`SEQAN_BEGIN_TESTSUITE(...)` and `SEQAN_END_TESTSUITE` are macros that expand to book-keeping code for running a test suite. `SEQAN_DEFINE_TEST(...)` expands to the definition of a function that runs a test.

Getting Started With Our Test Template

To make creating tests easier the code generator `util/bin/skel.py` has a command to generate test skeletons for you. As parameters, you give it the name of the module you want to test and the path to the repository. For example, use `skel.py tests my_module .` to create tests for the module *my_module* in the directory `tests`:

```
seqan $ ./util/bin/skel.py test my_module .
...
tests/my_module/
- CMakeLists.txt
- test_my_module.cpp
- test_my_module.h
```

Afterwards, you can compile and run the tests:

```
$ mkdir -p build/Debug
$ cd build/Debug
$ cmake ../..
$ make test_my_module
$ ./tests/my_module/test_my_module
...
```

Note: When adding new tests you have to add them to the dependencies of the test target in *tests/my_module/CMakeLists.txt*.

Test Macros

Inside your tests, you can use the `SEQAN_ASSERT*` and `SEQAN_ASSERT_*_MSG` macros to check for assertions. Other useful macros are SEQAN_PATH_TO_ROOT and SEQAN_TEMP_FILENAME.

The macros themselves are documented in the dox: `SeqAn API documentation AssertMacros`.

Assertion Caveats

When using one of the LT/GT/LEQ/GEQ/EQ/NEQ macros, the values have to provide a stream operator (`operator<<`) to write them to an output stream. If this is not implemented, then the assertion will not compile and something like the following will be printed by the compiler (in this case the GCC).

```
In file included from seqan/basic.h:55:0,
                 from tests/sequence/test_sequence.cpp:4:
seqan/basic/basic_testing.h: In function 'bool ClassTest::testEqual(const char*, int,␣
↪const T1&, const char*, const T2&, const char*, const char*, ...) [with T1 = Iter
↪<String<char, Block<3u> >, PositionIterator>, T2 = Iter<String<char, Block<3u> >,␣
↪PositionIterator>]':
seqan/basic/basic_testing.h:435:81:   instantiated from 'bool␣
↪ClassTest::testEqual(const char*, int, const T1&, const char*, const T2&, const␣
↪char*) [with T1 = Iter<String<char, Block<3u> >, PositionIterator>, T2 = Iter<String
↪<char, Block<3u> >, PositionIterator>]'
tests/sequence/test_string.h:386:2:   instantiated from 'void TestStringBasics()␣
↪[with TMe = String<char, Block<3u> >]'
tests/sequence/test_string.h:475:45:   instantiated from here
seqan/basic/basic_testing.h:385:13: error: no match for 'operator<<' in 'std::operator
↪<< [with _Traits = std::char_traits<char>](((std::ostream&
↪)((std::ostream*)std::operator<< [with _Traits = std::char_traits<char>
↪](((std::ostream&)((std::ostream*)std::operator<< [with _Traits = std::char_traits
↪<char>](((std::ostream&)((std::ostream*)std::operator<< [with _Traits = std::char_
↪traits<char>](((std::ostream&)((std::ostream*)std::operator<< [with _Traits =␣
↪std::char_traits<char>](((std::ostream&
↪)((std::ostream*)((std::ostream*)std::operator<< [with _Traits = std::char_traits
↪<char>](((std::ostream&)((std::ostream*)std::operator<< [with _Traits = std::char_
↪traits<char>](((std::ostream&)(& std::cerr)), file))), ((const char*)":")))->
↪std::basic_ostream<_CharT, _Traits>::operator<< [with _CharT = char, _Traits =␣
↪std::char_traits<char>](line))), ((const char*)" Assertion failed : "))))),␣
↪expression1))), ((const char*)" == "))))), expression2))), ((const char*)" was: ")) <
↪< value1'
```

The workaround is to use

```
SEQAN_ASSERT(end(str3) == begin(str3) + 7);
```

instead of

```
SEQAN_ASSERT_EQ(end(str3), begin(str3) + 7);
```

Best Practices

Rules are there to make you think before you break them. The following is not written into stone, but should be good guidelines. Improvements to the best practices is welcome.

Be Consistent

Whatever you do: Be consistent. If the one has read one part of your code then one should not have to adjust to different variable and function naming, comment style etc.

Tests Should Compile Without Warnings

Make sure that your tests compile without warnings. A common warning is "comparison of signed and unsigned integer".

In many places, the problematic line looks like this

```
SEQAN_ASSERT_LT(length(ndl), 30);
```

The `length` function returns an unsigned integer while the string literal `30` represents a (signed) `int`. You can fix this by changing the type of the number literal:

```
SEQAN_ASSERT_LT(length(ndl), 30u);
```

Break Your Tests Down

Each test should verify a part of the library as small as possible while still being meaningful. Having short test functions makes them easier to read and maintain.

Another advantage is that bogus state does not leak into other tests: imagine, you have a test that tests a function `assign_if_positive(a, b)` that assigns b to a if b is positive.

```
SEQAN_DEFINE_TEST(test_assign)
{
    int x = 0;

    assign_if_positive(x, 5);
    SEQAN_ASSERT_EQ(x, 5);

    assign_if_positive(x, -7);
    SEQAN_ASSERT_EQ(x, 5);
}
```

Now, what happens if `assign_if_positive(...)` has a bug and *never* assigns a value to its first parameter or always assigns 1? Both of your assertions will fail. This means you do not really know in which case the function works well and in which case it does not work well.

Splitting the test makes it more robust:

```
SEQAN_DEFINE_TEST(test_assign_positive)
{
    int x = 0;
    assign_if_positive(x, 5);
    SEQAN_ASSERT_EQ(x, 5);
}

SEQAN_DEFINE_TEST(test_assign_negative)
{
    int x = 0;
    assign_if_positive(x, -7);
    SEQAN_ASSERT_EQ(x, 0);
}
```

Use Helper Functions For Setup/TearDown

If you need to initialize the same state for multiple tests, then the code for this should only exist once. This makes it easier to maintain since we do not have to change it in multiple places at once. This is especially useful when following the best practice *Break Your Tests Down*.

Example:

Instead of

```
SEQAN_DEFINE_TEST(test_grep)
{
    char *contents = loadFile("corpus.txt");

    int pos = doGrep(contents, "nonexisting pattern");
    SEQAN_ASSERT_EQ(pos, -1);

    pos = doGrep(contents, "existing pattern");
    SEQAN_ASSERT_EQ(pos, 3);

    delete contents;
}
```

do

```
// Set-up for test_grep_{success, failure}.
void testGrepSetUp(const char *filename, char *outContents)
{
    outContents = loadFile(filename);
}

// Tear-down for test_grep_{success, failure}.
void testGraphTearDown(char *contents)
{
    delete contents;
}

// Test greping for existing patterns.
SEQAN_DEFINE_TEST(test_grep_success)
{
    // corpus.txt contains the string "1234existing pattern567".
    char *contents;
    testGrepSetUp("corpus.txt", contents);

    int pos = doGrep(contents, "existing pattern");
    SEQAN_ASSERT_EQ(pos, 3);

    testGrepTearDown(contents);
}

// Test greping for non-existing patterns.
SEQAN_DEFINE_TEST(test_grep_failure)
{
    // corpus.txt contains the string "1234existing pattern567".
    char *contents;
    testGrepSetUp("corpus.txt", contents);

    int pos = doGrep(contents, "nonexisting pattern");
    SEQAN_ASSERT_EQ(pos, -1);
```

```
    testGrepTearDown(contents);
}
```

Comment Your Tests

Tests can complement examples from the documentation in that they illustrate each call to your code's API. Thus, make sure that your tests are well-documented. Not only for users who look up how to use your code but also for the next maintainer.

There should be a documentation of the test itself and also inline comments. In your comments, you should focus on the maintainer and not so much on the user. Even if some things are obvious, you might want to illustrate why you call a function with the given parameters, e.g. describe the corner cases.

Example:

```
// Test abs() function with 1, a representative for positive values.
SEQAN_DEFINE_TEST(test_abs_with_one)
{
    SEQAN_ASSERT_EQ(abs(1), 1);
}

// Test abs() function with 0, the only corner case here.
SEQAN_DEFINE_TEST(test_abs_with_zero)
{
    SEQAN_ASSERT_EQ(abs(0), 0);
}

// Test abs() function with -1, a representative for negative values.
SEQAN_DEFINE_TEST(test_abs_with_minus_one)
{
    SEQAN_ASSERT_EQ(abs(-1), 1);
}
```

ToC

Contents

Writing App Tests

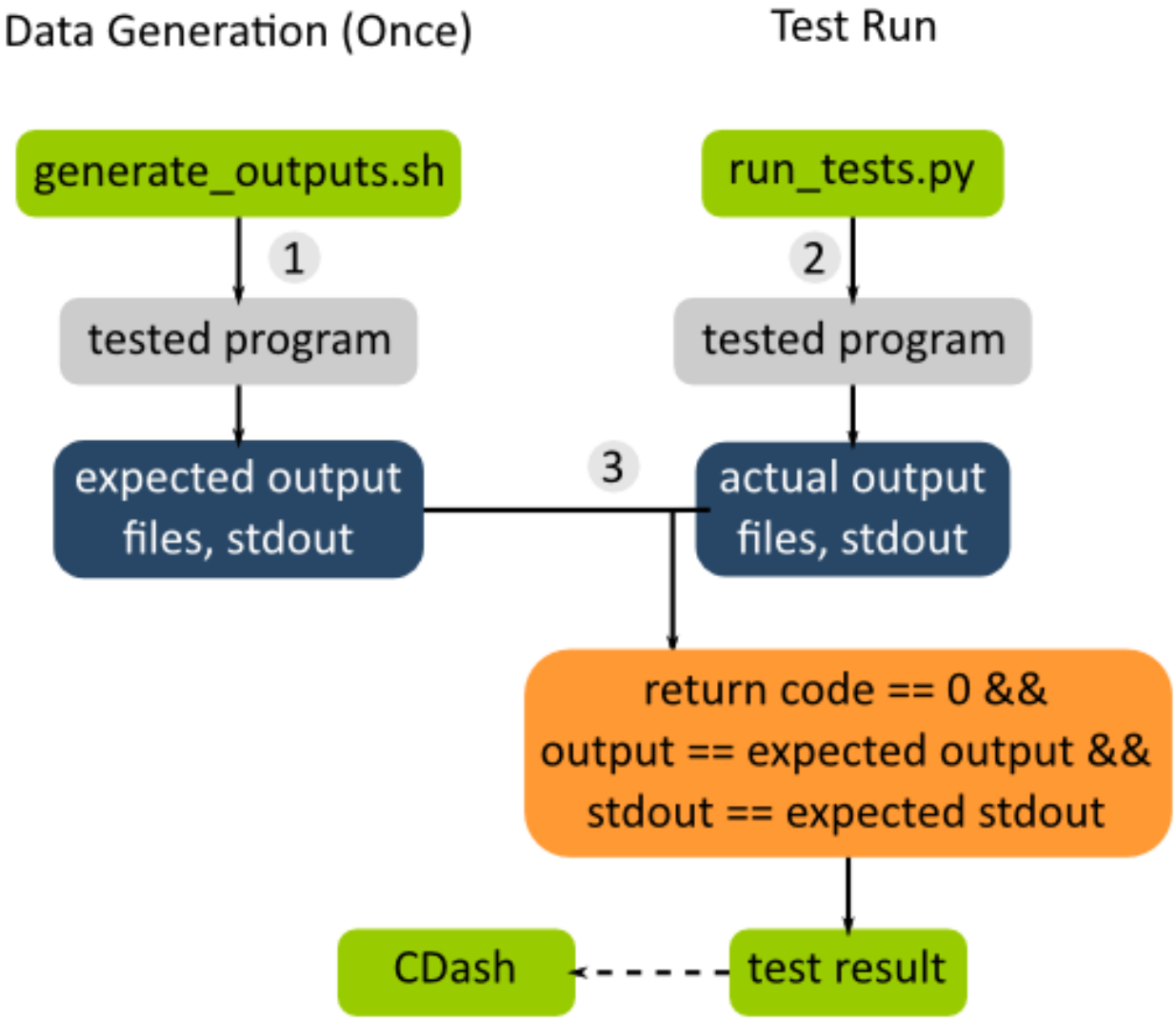
This HowTo describes the basics of writing your own app tests.

Caution: SeqAn does not accept new apps into the repository so this document is less relevant for the future.

Overview

SeqAn application tests allow the simple testing of apps: the application is run several times, each with a different set of parameters and/or input files. The output to STDOUT and STDERR is captured. The captured output and all files written by the application are then compared against "gold standard" output files that were previously generated.

The general data flow for app tests is as follows (a simple working example are the app tests for the app **pair_align**).



There are two major steps involved: (A) Test data generation and (B) actually running the tests.

Test Data Generation

This is done once when preparing the test.

The program is run with different parameters (in the case of pair_align with different algorithms, scores, alphabets etc) and input files (for pair_align, e.g. for proteins and DNA sequences). The output files and the output to stdout (logs) are collected. The output and logs are then stored as files in the SVN repository and serve as the reference for future comparisons.

The expected output files are mostly generated by running a "sane" version of the program and then being checked for problems. Since there are a lot, they are usually generated by shell files, the *generate_outputs.sh* files. These files also serve as documentation which settings were used for which output files. Generally, they mirror the structure of the app test Python script (described below).

To reiterate, the shell script is usually only executed once when the tests are created. The output needs to be regenerated only when the program output changes and this change is **deliberate**. They are *not* regenerated on each test run.

Note that the path to the binary that is used to generate the output should be stored in a shell variable at the top of the file. This allows for easily changing this. These shell scripts should also be well-documented. See the *generate_outputs.sh* file of pair_align or tree_recon for simple and mostly clean examples.

Running Tests

The app tests are then run in the nightly CMake builds and their results are submitted to CDash. There are two steps involved here: (1) Executing the programs and (2) comparing their result with the expected ones. There is a Python test driver program (called *run_tests.py* by convention) for each collection of app tests.

These programs us the Python module *seqan.app_tests* for running and usually mirror the corresponding *generate_outputs.sh* file.

Creating App Tests

We will create app tests for a small app that converts its argument to upper case and prints it to stdout.

Setup App "upcase"

First, create the app using *skel.py*.

```
$ ./util/bin/skel.py app upcase .
```

Then, edit *upcase.cpp* to look as follows:

```
#include <iostream>
#include <cstring>

using namespace seqan;

int main(int argc, char const ** argv)
{
    if (argc <= 1)
    {
        std::cerr << "No arguments given!" << std::endl;
        return 1;
    }

    for (int i = 1; i < argc; ++i)
    {
        for (char const * ptr = &argv[i][0]; *ptr != '\0'; ++ptr)
            std::cout << static_cast<char>(toupper(*ptr));
```

```
        std::cout << std::endl;
    }

    return 0;
}
```

Then, go to your build directory (here: *build/Debug*), build the app, and make sure it works correctly.

```
$ cd build/Debug
$ cmake .
$ cd apps/upcase
$ make
$ ./upcase This is a test
THIS
IS
A
TEST
```

Creating App Tests

You can use the *skel.py* program to create the app tests.

```
$ cd ../../../../../..
$ ./util/bin/skel.py app_tests apps/upcase/
```

As suggested by the output of *skel.py*, add the following to your *apps/upcase/CMakeLists.txt*:

```
# Add app tests if Python interpreter could be found.
if(PYTHONINTERP_FOUND)
  add_test(NAME app_test_upcase COMMAND ${PYTHON_EXECUTABLE}
    ${CMAKE_CURRENT_SOURCE_DIR}/tests/run_tests.py ${CMAKE_SOURCE_DIR}
    ${CMAKE_BINARY_DIR})
endif(PYTHONINTERP_FOUND)
```

Now, open the file *apps/upcase/tests/generate_outputs.sh* and modify it as follows.

```
#!/bin/sh
#
# Output generation script for upcase

UPCASE=../../../../../build/Debug/apps/upcase/upcase

# ============================================================
# Generate Output
# ============================================================

${UPCASE} simple example > simple.stdout
${UPCASE} 'another()/' 'examplE!' > other.stdout
```

We now run the program two times with different arguments and stored the output in files *simple.stdout* and *other.stdout*. These files are kept in the directory *apps/upcase/tests* and can now go into version control.

```
$ cd apps/upcase/tests
$ ./generate_outputs.sh
$ head -1000 simple.stdout other.stdout
===> simple.stdout <===
```

```
SIMPLE
EXAMPLE

===> other.stdout <===
ANOTHER()/
EXAMPLE!
```

Now, we have the expected test output files. We now have to modify the test driver script *run_tests.py*. Open the file *apps/upcase/tests/run_tests.py*. This file is a Python script that runs the programs, collects their output and compares the expected output prepared above with the actual one. It should look like the following:

```
#!/usr/bin/env python2
"""Execute the tests for upcase.

The golden test outputs are generated by the script generate_outputs.sh.

You have to give the root paths to the source and the binaries as arguments to
the program.  These are the paths to the directory that contains the 'projects'
directory.

Usage:  run_tests.py SOURCE_ROOT_PATH BINARY_ROOT_PATH
"""
import logging
import os.path
import sys

# Automagically add util/py_lib to PYTHONPATH environment variable.
path = os.path.abspath(os.path.join(os.path.dirname(<u>file</u>), '..', '..',
                                    '..', '..', 'util', 'py_lib'))
sys.path.insert(0, path)

import seqan.app_tests as app_tests

def main(source_base, binary_base):
    """Main entry point of the script."""

    print 'Executing test for upcase'
    print '========================='
    print

    ph = app_tests.TestPathHelper(
        source_base, binary_base,
        'apps/upcase/tests')  # tests dir

    # ============================================================
    # Auto-detect the binary path.
    # ============================================================

    path_to_program = app_tests.autolocateBinary(
      binary_base, 'apps/upcase', 'upcase')

    # ============================================================
    # Built TestConf list.
    # ============================================================

    # Build list with TestConf objects, analogously to how the output
    # was generated in generate_outputs.sh.
    conf_list = []
```

```
    # ============================================================
    # First Section.
    # ============================================================

    # App TestConf objects to conf_list, just like this for each
    # test you want to run.
    conf = app_tests.TestConf(
        program=path_to_program,
        redir_stdout=ph.outFile('STDOUT_FILE'),
        args=['ARGS', 'MUST', 'BE', 'STRINGS', str(1), str(99),
              ph.inFile('INPUT_FILE1'),
              ph.inFile('INPUT_FILE2')],
        to_diff=[(ph.inFile('STDOUT_FILE'),
                  ph.outFile('STDOUT_FILE')),
                 (ph.inFile('INPUT_FILE1'),
                  ph.outFile('INPUT_FILE1'))])
    conf_list.append(conf)

    # ============================================================
    # Execute the tests.
    # ============================================================
    failures = 0
    for conf in conf_list:
        res = app_tests.runTest(conf)
        # Output to the user.
        print ' '.join(['upcase'] + conf.args),
        if res:
            print 'OK'
        else:
            failures += 1
            print 'FAILED'

    print '=============================='
    print '     total tests: %d' % len(conf_list)
    print '    failed tests: %d' % failures
    print 'successful tests: %d' % (len(conf_list) - failures)
    print '=============================='

    # Compute and return return code.
    return failures != 0


if <u>name</u> == '<u>main</u>':
    sys.exit(app_tests.main(main))
```

Here, we now mirror the *generate_outputs.sh* file by replacing the section *First Section* with the following:

```
# ============================================================
# Run Program upcase.
# ============================================================

# Simple Example.
conf = app_tests.TestConf(
    program=path_to_program,
    redir_stdout=ph.outFile('simple.stdout'),
    args=['simple', 'example'],
    to_diff=[(ph.inFile('simple.stdout'),
```

```
              ph.outFile('simple.stdout'))])
conf_list.append(conf)

# Another Example.
conf = app_tests.TestConf(
    program=path_to_program,
    redir_stdout=ph.outFile('other.stdout'),
    args=['another()/', 'examplE!'],
    to_diff=[(ph.inFile('other.stdout'),
              ph.outFile('other.stdout'))])
conf_list.append(conf)
```

Finally, we can run the program using ctest.

```
$ cd ../../../../..
$ cd build/Debug/apps/upcase
$ ctest .
```

If everything goes well, the output will be as follows:

```
$ ctest .
Test project ${PATH_TO_CHECKOUT}/build/Debug/apps/upcase
    Start 1: app_test_upcase
1/1 Test #1: app_test_upcase ..................   Passed    0.04 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.05 sec
```

In the case of failures, the output could be as follows.

```
$ ctest .
Test project /home/holtgrew/Development/seqan-trunk/build/Debug/apps/upcase
    Start 1: app_test_upcase
1/1 Test #1: app_test_upcase ..................***Failed    0.02 sec

0% tests passed, 1 tests failed out of 1

Total Test time (real) =   0.03 sec

The following tests FAILED:
      1 - app_test_upcase (Failed)
Errors while running CTest
```

The `ctest` command has many options. A useful one for debugging is `--output-on-failure` which will print the test result if the test does not succeed. For example, the output could be as follows:

```
$ ctest . --output-on-failure
Test project /home/holtgrew/Development/seqan-src/build/Debug/apps/upcase
    Start 1: app_test_upcase
1/1 Test #1: app_test_upcase ..................***Failed    0.02 sec
Traceback (most recent call last):
  File "/home/holtgrew/Development/seqan-trunk/apps/upcase/tests/run_tests.py", line␣
↪16, in <module>
    import seqan.app_tests as app_tests
ImportError: No module named seqan.app_tests

0% tests passed, 1 tests failed out of 1
```

```
Total Test time (real) =   0.03 sec

The following tests FAILED:
      1 - app_test_upcase (Failed)
Errors while running CTest
```

This is a common error that tells us that we have to appropriately set the environment variable *PYTHONPATH* so the module *seqan.app_tests* is available from within Python.

You have to add `util/py_lib` to your PYTHONPATH. On Linux/Mac Os X, you can do this as follows (on Windows you have to set the environment variable, e.g. following this guide).

```
$ export PYTHONPATH=${PYTHONPATH}:PATH_TO_SEQAN/util/py_lib
```

Now, you test should run through:

```
$ ctest . --output-on-failure
...
```

ToC

Contents

Writing Tutorials

At the bottom, you can find a *Tutorial Template* for starting a new tutorial.

Conventions

Wiki Conventions

- Use only one line per sentence. This increases the readability of the sources.

Naming Conventions

- Use headline capitalization for headlines.
- Use the tutorial's title as the file name (e.g. `/wiki/Tutorial/NameOfYourTutorial.rst`).
- Assignments are numbered in the order they appear in a tutorial (e.g. `Assignment 5`). Do not use a section relative numbering but an absolute one. If, e.g., the last assignment of section 1 was assignment 3, the first assignment of section 2 is assignment 4).
- Place the assignment's solutions inline.

Design & Layout Conventions

- Use back ticks (``` `` ```) to denote names of variables, functions, etc. (e.g. ``` ``append`` ``` results in `append`).
- Use bold font (`**word**`) to denote key concepts.
- Use `item` and `menu > sub menu > item` to denote GUI entries and menu paths.
- Use the following markup to include source code

```
.. includefrags:: demos/tutorial/alignment/alignment_msa.cpp
   :fragment: init
```

  where `demos/tutorial/tutorial/alignment/alignment_msa.cpp` gives the source code file in the repository and `init` the fragment to include in the tutorial.

- You should always build and test the tutorials code snippets before using them.

```
manual # make html
```

- Use the following markup to format screen output:

```
::

    # Hello World!
```

- Use the following markup to inform about **important bugs** or other relevant issues. The content (and thereby the box itself) is always of **temporary** nature and should **only be used thriftily**.:

```
.. warning::

   Warning goes here.
```

- Use the following markup to give **important information**.

  These boxes contain information that **should be kept in mind** since the described phenomenon is very likely to be encountered by the reader again and again when working with SeqAn. In contrast to the `.. warning::`, this box type is of **permanent** nature and the given information are valid for a long time.

```
.. important::

   Important information goes here...
```

Use the following markup to give further / **optional information**. These are information that support the understanding but are too distinct to be put in a normal paragraph.:

```
.. hint::

   Optional information goes here.
```

- Use the following markup to format assignments (for further details see *Assignments*):

```
.. container:: assignment

   The assignment goes here.
```

- Use `:dox:`DocItem`` to create links to the SeqAn API dox documentation.

Important: Note that this will mereley generate the URLs that **dddoc** would create but does not perform any checking. Some examples:

  - String (`:dox:`String``)
  - AllocString (`:dox:`AllocString``)
  - Alloc String (`:dox:`AllocString Alloc String``)
  - StringConcept (`:dox:`StringConcept``)

Structure

Meta Information

Place the directives for the side bar and the link target for the tutorial page directly before the tutorial title.

```
.. sidebar:: ToC

   .. contents::


.. _tutorial-datastructures-sequences:

Sequences
---------
```

Based on the *Tutorial Template*, provide information regarding:

learning objective Describe the learning objective in your own words.

difficulty Valid values: Very basic, Basic, Average, Advanced, Very advanced

duration In average how much time will a user spend on absolving this tutorial? If you expect more than 90 minutes please split your tutorial up into multiple ones.

prerequisites A list of absolved tutorials and other requirements you expect your reader to fulfill.

Introduction

In the next paragraph introductory information are given that answer the following questions:

- What is this tutorial about?
- Why are the information important?
- What are the communicated information used for?
- What can the reader expect to know after having absolved the tutorial?

Section

Introduction

In each section's introduction part you answer the following questions:

- What is this section about?
- What are the central concepts in this section?
- What is your partial learning objective?

Explanations / Examples

The main part consists of the description of the topic. This is the space where enough knowledge is transmitted to **enable the reader to solve all assignments**. Further details are contained in the *Tutorial Template* and in the didactics section.

Try not to get lost in details. If you have useful but still optional information to give use a `.. note::` directive.

Assignments

The assignments' purpose in general is to support the reader's understanding of the topic in question. For this each assignment is of a special type (Review, Application and Transfer), has an objective, hints and a link to the complete solution.

Depending on the type of assignment the reader is guided through the assignment solving by providing him with partial solutions.

There must always be an assignments of type Review. Assignments must always appear in an ascending order concerning their types and no "type gap" must occur.

Thus the only valid orders are:

- Review
- Review, application
- Review, application, transfer

The order Review, transfer is invalid since a "type gap" (application type missing) occurred.

All assignments must be accompanied by a solution.

Further Section

as many further sections as you like

Didactics

Type

As already mentioned in the assignment structure description each assignment is of one type.

These levels are

Review knowledge fortification (mainly through repetition, optionally with slight variations)

Application supervised problem solving (finely grained step-by-step assignment with at least one hint and the interim solution per step)

Transfer knowledge transfer (problem solving in a related problem domain / class)

Based on the chosen level you should design your assignment.

Duration

The time needed to absolve a tutorial must not exceed 90 minutes. Split your tutorial up (e.g. Tutorial I, Tutorial II) if you want to provide more information.

Language

Make use of a simple language. This is neither about academic decadence nor about increasing the learning barrier. You are not forced to over-simplify your subject but still try to use a language that is also appropriate for those who don't fully meet the tutorials prerequisites.

Mental Model

When your describe and explain your topic give as many examples as possible. Try to adopt the reader's perspective and imagine - based on your target group and prerequisites - your reader's mental model. The mental model can be described as an imagination of the interaction of central concepts. Try to support the reader in developing a mental model that fits best to your topic.

Integration

- Add a link to your tutorial to `Tutorial.rst` and add a link to the `.. toctree`.
- Above you stated the tutorials your tutorial has as prerequisites. Add the link in a way that all required tutorials are listed above your tutorial.

Tutorial Template

```
.. sidebar:: ToC

   .. contents::


.. _tutorial-tutorial-template:

Tutorial Template
```

```
=================

Learning Objective
  Describe the learning objective in your own words.
  **Example:**
  You will be able to write a tutorial that meets our quality standards.

Difficulty
  [Very basic, Basic, Average, Advanced, Very advanced]
  **Example:**
  Basic

Duration
  In average how much time will a user spend on absolving this tutorial?
  If you expect more than 90 minutes please **split your tutorial up** into multiple␣
↪ones.
  **Example:**
  1 h

Prerequisites
  A list of absolved tutorials and other requirements you expect your reader to␣
↪fulfill.
  **Example:** :ref:`tutorial-getting-started-first-steps-in-seqan`, :ref:`tutorial-
↪algorithms-pattern-matching`, English language

This is the place where introductory need to be in given, e.g. "This page constitutes␣
↪the template for all future SeqAn tutorials".

Use this and optional further paragraphs to give the following information:

* What is this tutorial about?
* Why are the information important?
* What are the communicated information used for?
* What can the reader expect to know after having absolved the tutorial?

.. warning::

   This is a warning message.

   Here you can inform users about important bugs or other relevant issues.

Section
-------

Use this and optional further paragraphs to give the following information:

* What is this section about?
* What are the central concepts in this section?
* What is your partial learning objective?

When your describe and explain your topic give **as many examples as possible**.
Try to adopt the reader's perspective and imagine - based on your target group and␣
↪prerequisites - your **reader's mental model**.
The mental model can be described as an imagination of the interaction of central␣
↪concepts.
Use a **simple language** and try to support the reader in developing a mental model␣
↪that fits best to your topic.
```

```
.. tip::

   What are tips for?

   An ``.. tip`` ist useful to give information that are **optional** and thus don't␣
↪need to be read.
   Typical information are **further details** that support the understanding but are␣
↪too distinct to be put in a normal paragraph.

   In this example you could tell the reader more about didactics and give him some␣
↪useful links.

.. important::

   What are important blocks for?

   These boxes contain information that **should be kept in mind** since the␣
↪described phenomenon is very likely to be encountered by the reader again and again␣
↪when working with SeqAn.

Subsection
^^^^^^^^^^

If you give code examples tell the reader what he can see and what is crucial to your␣
↪snippet.
Link all classes and other resources to the SeqAn documentation system by using␣
↪``:dox:Item` (e.g. :dox:`String`).
In order to include code snippets use ``.. includefrags:: path``.

.. includefrags:: demos/tutorial/alignments/alignment_banded.cpp
   :fragment: alignment

If possible also include the generated output by given code in the console.
Here is one example:

.. code-block:: console

   0: ACAG
   1: AGCC
   2: CCAG
   3: GCAG
   4: TCAG

Now that you gave an overview of important concepts of your topic let the user play␣
↪with it!
Formulate **small assignments** to allow the reader to fortify his newly acquainted␣
↪knowledge.

Assignment 1
""""""""""""

.. container:: assignment

   Type
     [Review, Application, Transfer]

     Note that your readers will be in different phases of learning. For the sake of␣
↪simplicity we restrict ourselves to the following three levels:
```

```
    #. knowledge fortification (mainly through repetition, optionally with slight␣
↪variations)
    #. supervised problem solving (finely grained step-by-step assignment with at␣
↪least one hint and the interim solution per step)
    #. knowledge transfer (problem solving in a related problem domain / class)

    **Example:** Application

  Objective
    The objective of the assignment.
    **Example:**
    Output all symbols a given alphabet can have.
    The output should look like this: ...

  Hints
    ...

  Solution
    .. container:: foldable

       Foldable solution with description.

    This part of the assignment is to give partial solutions.
    A partial solution starts with a sentence of what this step is about and gives␣
↪the lines of code that are needed to implement this step.

    Solution Step 1
      .. container:: foldable
        The given sequence are of alphabet...
        Therefore, you have to...

        .. includefrags:: demos/tutorial/alignments/alignment_banded.cpp
           :fragment: main

    Solution Step 2
      .. container:: foldable
        The given sequence are of alphabet...
        Therefore, you have to...

        .. includefrags:: demos/tutorial/alignments/alignment_banded.cpp
           :fragment: fragment
```

ToC

Contents

Library and App releases

There are three different "packaging targets":

1. a source package of the SeqAn library (`-DSEQAN_BUILD_SYSTEM=SEQAN_RELEASE_LIBRARY`)
2. a package containing all apps (`-DSEQAN_BUILD_SYSTEM=SEQAN_RELEASE_APPS`)
3. a package containing a single SeqAn app (`-DSEQAN_BUILD_SYSTEM=APP:$appname`)

We assume that you have read *Installing SeqAn* and have cloned or unzipped the **full SeqAn sources** to `~/devel/seqan` (not the "library sources" described in other places).

The instructions for all packaging targets are the same (replace `$pack_target` with the above string):

```
~ # mkdir -p ~/devel/seqan-build/deploy
~ # cd ~/devel/seqan-build/deploy
deploy # cmake ../../seqan -DSEQAN_BUILD_SYSTEM=$pack_target -DSEQAN_STATIC_APPS=1 -
↪DSEQAN_ARCH_SSE4=1
deploy # make package
```

On Windows, replace the last command with

```
deploy # cmake --build . --target PACKAGE
```

Depending on the platform this might create a ZIP-file, a tarball and/or a platform specific installer.

Official Packages

We provide (1) a source package of SeqAn library; and for each official application (3) single binary packages for different operating systems and architectures.

Note: Especially when creating packages, make sure that the cmake generator and/or compiler are the ones you want!

GNU/Linux, macOS & BSD

- The binary packages should be built on the **oldest supported kernel** and with the **oldest supported GCC** compiler.
- The CMake version on the building system should be at least 3.1.
- Builds should be static (`-DSEQAN_STATIC_APPS=1`).
- There should be a 32Bit package, built on a 32Bit system or cross-compiled (`-DCMAKE_CXX_FLAGS="-m32"`).
- There should be a 64Bit package.
- There should be an optimized 64Bit build (`-DSEQAN_ARCH_SSE4=1`).
- For applications where it makes sense, a further optimized build *can* be provided (`-DSEQAN_ARCH_AVX2=1`)

Windows

- The binary packages should be built with the latest **Intel C++ Compiler** for performance and compatibility reasons (see *here*).

- There should be a 32Bit package, built on a 32Bit system or cross-compiled (see *here*).
- There should be a 64Bit package.

Downstream Packaging

These are some guidelines for creating SeqAn packages for operating system specific packaging systems, like *apt* (Debian/Ubuntu) or *rpm* (Fedora/RedHat/CentOS/SUSE).

Library Package

We recommend that downstream package maintainers provide one package named **seqan** that contains only the header-library and the api-docs and that is built from our *library packages* available here: http://packages.seqan.de

They have the advantage of not requiring any build steps, simply copy the `include` and `share` directories to the desired locations.

Application Package(s)

Beyond that package maintainers have the choice to create either a single package called **seqan-apps** that contains all the applications *or* a seperate package per application (with the respective name of that app). Based on the above instructions this should be fairly easy to accomplish.

ToC

Contents

Nightly Builds

Every night the master and develop branches of SeqAn are fetched and built on a variety of platforms. The results can be seen at the SeqAn CDash site.

The scripts that facilitate this are hosted here. Please note that the `linux` and `macosx` directories are outdated, both are now handled by the `unix` directory.

Unix Script variables

Variable	Description
BITS	32 or 64 (64 by default)
GIT_BRANCH	master, develop or a valid branch name (develop by default)
COMPILERS	list of compiler-binaries to use
COMPILER_FLAGS	flags to append to the compiler calls
WITH_MEMCHECKS	if set to anything but 0 CTEST will perform memchecks
WITH_COVERAGE	if set to anything but 0 CTEST will perform coverage checks
MODEL	Nightly, Experimental or Continuous (defaults to Experimental); this only influences the section where it is printed in CDash
TMPDIR	place to store temporary files of run (will be pruned after run; defaults to /tmp)
TESTROOT	The place checkouts and builds take place (if unset defaults to TMPDIR, which means it will be pruned; otherwise it will be reused on next run)
THREADS	number of threads to use (defaults to 1)

Please see the up-to-date variables here.

Unix cron jobs

To setup the build, checkout the subversion directory mentioned above and decide on the variables you wish to set. Remember to give TMPDIR and TESTROOT enough space.

Then open your crontab with

```
crontab -e
```

And add the jobs that you wish to have executed. It could look like this:

```
# Shell variable for cron
SHELL=/bin/sh
# PATH variable for cron
PATH=/usr/local/libexec/ccache:/usr/local/bin:/usr/local/sbin:/sbin:/usr/sbin:/bin:/
↪usr/bin
#m h d m w
5  1 * * * MODEL=Nightly TMPDIR=/tmp TESTROOT=${HOME}/nightly-builds/testroot GIT_
↪BRANCH=master  BITS=32 COMPILERS="clang++35 clang++36 clang++37 clang++38 clang++-
↪devel" THREADS=4 nice -n 10 ${HOME}/nightly-builds/unix/bin/run.sh >/dev/null
5  1 * * * MODEL=Nightly TMPDIR=/tmp TESTROOT=${HOME}/nightly-builds/testroot GIT_
↪BRANCH=develop BITS=32 COMPILERS="clang++35 clang++36 clang++37 clang++38 clang++-
↪devel" THREADS=4 nice -n 10 ${HOME}/nightly-builds/unix/bin/run.sh >/dev/null

5  3 * * * MODEL=Nightly TMPDIR=/tmp TESTROOT=${HOME}/nightly-builds/testroot GIT_
↪BRANCH=master  BITS=64 COMPILERS="clang++35 clang++36 clang++37 clang++38 clang++-
↪devel g++49 g++5 g++6" THREADS=4 nice -n 10 ${HOME}/nightly-builds/unix/bin/run.sh >
↪/dev/null
5  3 * * * MODEL=Nightly TMPDIR=/tmp TESTROOT=${HOME}/nightly-builds/testroot GIT_
↪BRANCH=develop BITS=64 COMPILERS="clang++35 clang++36 clang++37 clang++38 clang++-
↪devel g++49 g++5 g++6" THREADS=4 nice -n 10 ${HOME}/nightly-builds/unix/bin/run.sh >
↪/dev/null
```

The first columns mean that on the 5th minute of the 1st/3rd hour (1:05am/3:05am) of every day, of every month and every week the subsequent command is executed. The variables are described above.

nice -n 10 ensures that the cron jobs get a low priority so the system remains responsive. The path after that is the path to your svn checkout. The redirection in the end prevents output from spamming your mail account.

Remember to add all folders to the PATH variable that contain binaries which you have added to `COMPILERS=`.

Windows

TODO double check this.

Now, get the build scripts:

```
copy seqan-src\misc\ctest\run_nightly.sh .
copy seqan-src\misc\ctest\Seqan_Nightly.cmake.example Seqan_Nightly.cmake
copy seqan-src\util\cmake\CTestConfig.cmake seqan-src\
```

Adjust the build name and site name in `Seqan_Nightly.cmake`. Now, test the setup by running:

```
run_nightly.bat
```

Add `run_nightly.bat` to nightly Scheduled Tasks of Windows (analogously to the CTest Tutorial):

1. Open `Scheduled Tasks` from Control Panel.
2. Select `Add Scheduled Task`.
3. Select `Next` to select command.
4. Click **Browse...** and select `run_nightly.bat`.
5. Click **Next** and select name and repetition date. Repetition date for Nightly dashboards should be `Daily`.
6. Click **Next** and select time to start the dashboard.
7. Click **Next** and select `Open advanced properties...` to fine tune the scheduled task.
8. Select **Next** and type password of the user.‘‘
9. Task is created. The Advanced Properties dialog should open.
10. In advanced properties, specify full command name. It is very important that you use double quotes in case you have spaces in your path.
11. Select `Ok`, which will ask for password again.
12. The new task should be created.

Miscellaneous Guides

Here are some general guides on programming and debugging that might be helpful to you when working with or on SeqAn.

ToC

Contents

Fixing Whitespace Automatically

This page describes how to use Universal Indent GUI and Uncrustify to automatically fix whitespace such that code resembles the *C++ Code Style* more closely.

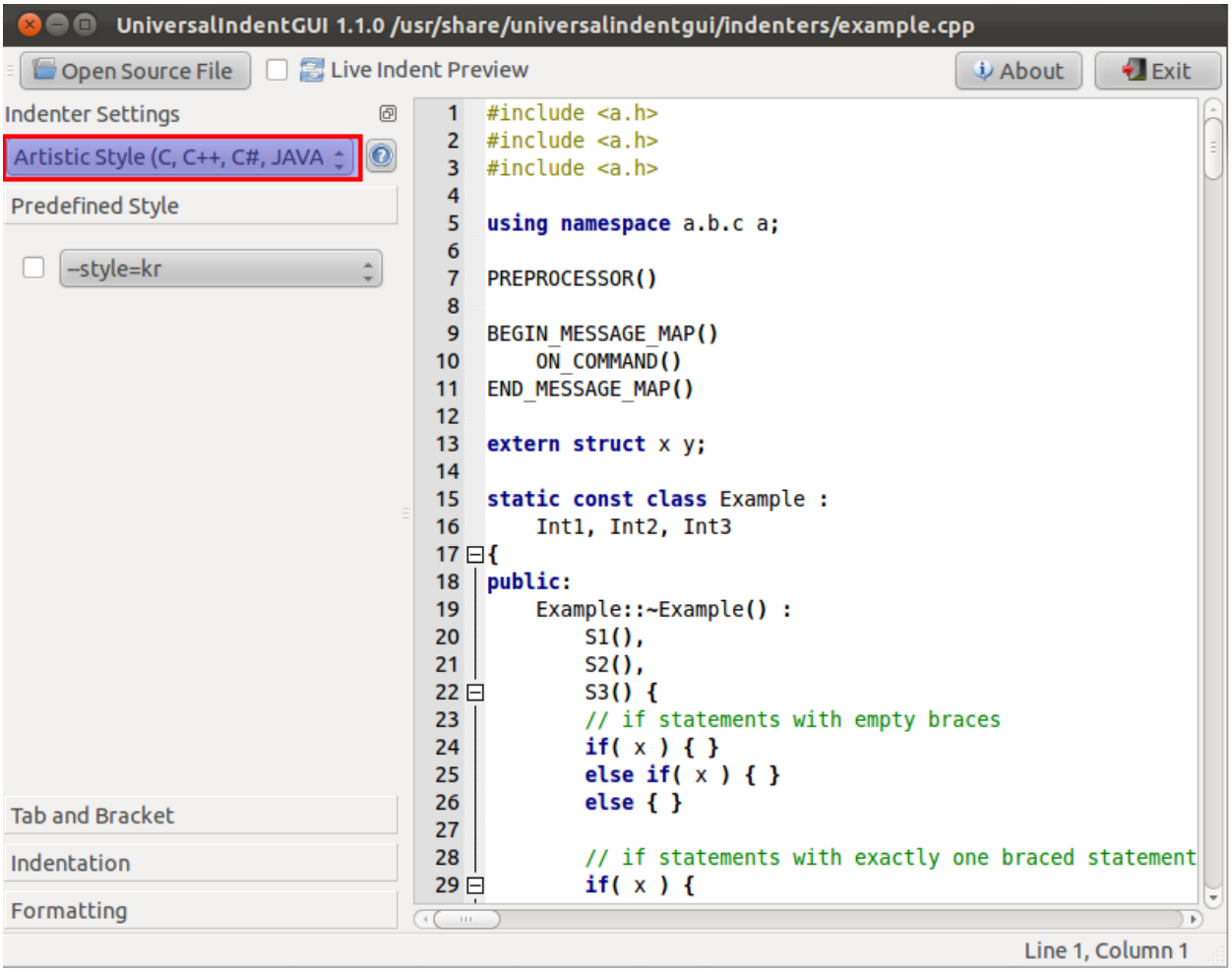
- Uncrustify is a command line program that is given a style definition and a source file and reformats the source file according to the configuration.
- Universal Indent GUI is a graphical front-end to Uncrustify with life preview that allows to manipulate the configuration and immediately see the results.

Installing Universal Indent GUI

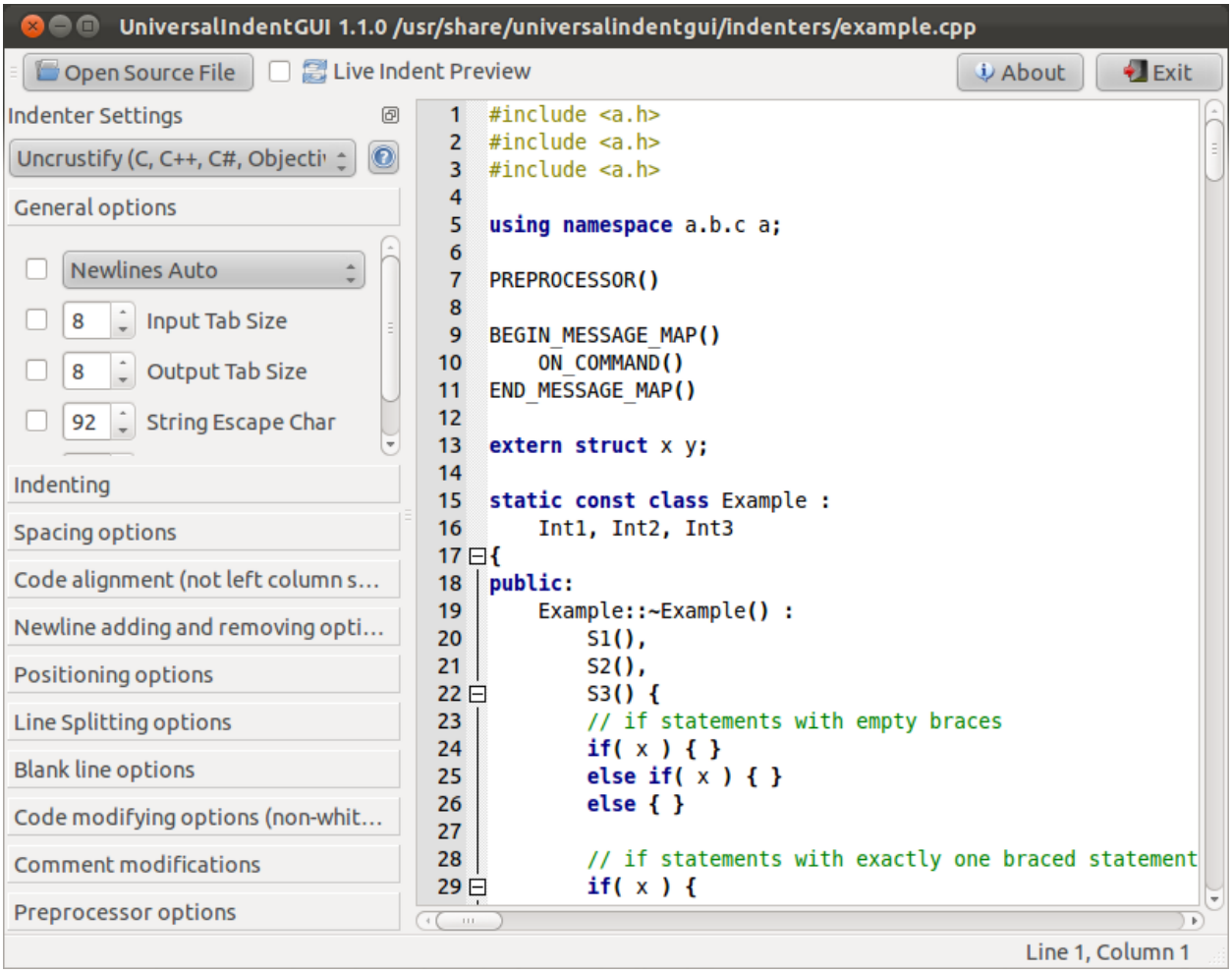
This one is pretty easy. On Ubuntu and other Linux systems, you can use the package management system to install the GUI and the reformatting programs. The Universal Indent GUI download page has binaries for Mac Os X and Windows.

Preview with Universal Indent GUI

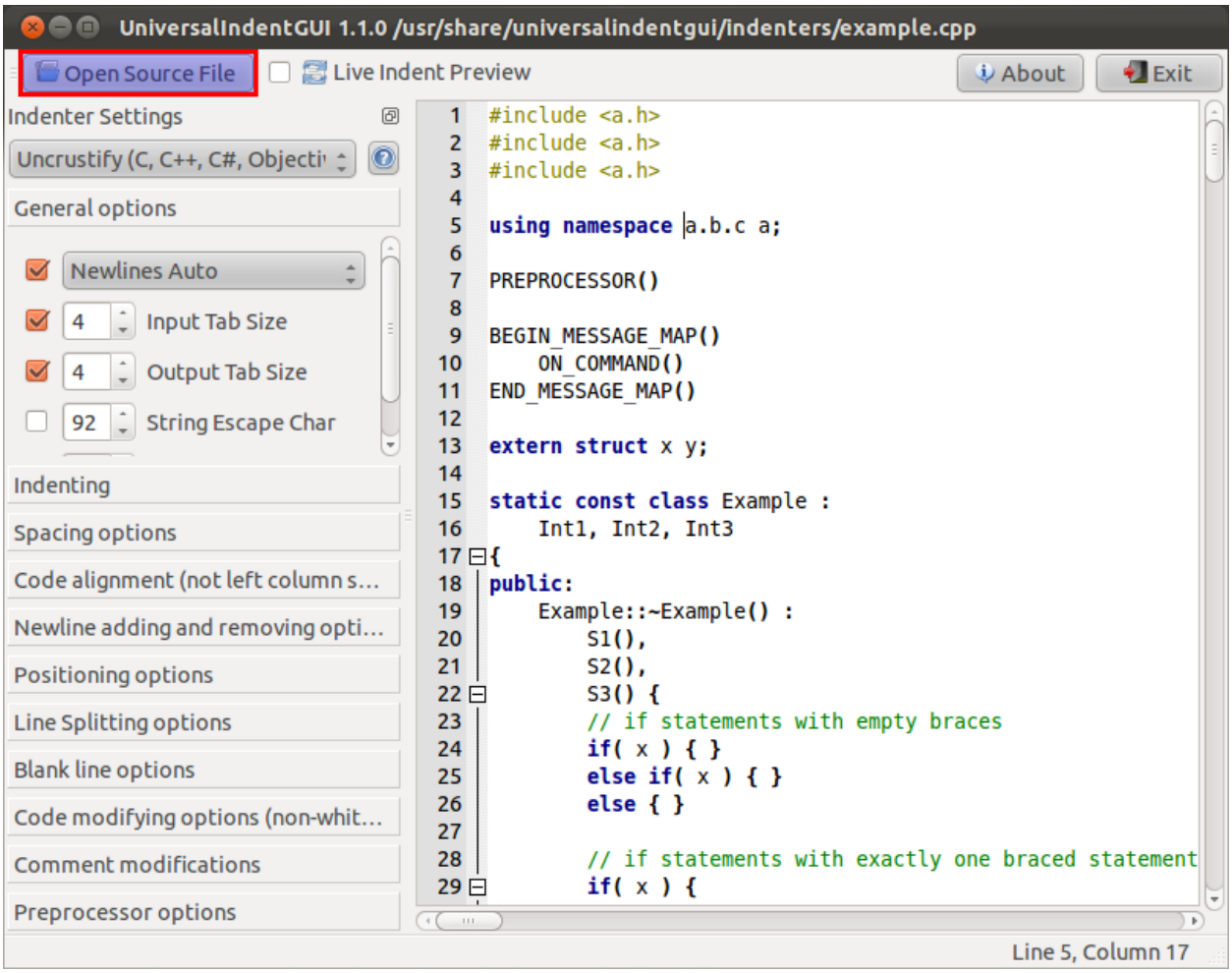
When started, the program will present you with a window like the following.



First, we set the indenter to Uncrustify.


UniversalIndentGUI 1.1.0 /usr/share/universalindentgui/indenters/example.cpp
Open Source File
Live Indent Preview
About
Exit
Indenter Settings
Uncrustify (C, C++, C#, Objecti
General options
Newlines Auto
8
Input Tab Size
8
Output Tab Size
92
String Escape Char
Indenting
Spacing options
Code alignment (not left column s...
Newline adding and removing opti...
Positioning options
Line Splitting options
Blank line options
Code modifying options (non-whit...
Comment modifications
Preprocessor options
1 #include <a.h>
2 #include <a.h>
3 #include <a.h>
4
5 using namespace a.b.c a;
6
7 PREPROCESSOR()
8
9 BEGIN_MESSAGE_MAP()
10 ON_COMMAND()
11 END_MESSAGE_MAP()
12
13 extern struct x y;
14
15 static const class Example :
16 Int1, Int2, Int3
17 {
18 public:
19 Example::~Example() :
20 S1(),
21 S2(),
22 S3() {
23 // if statements with empty braces
24 if(x) { }
25 else if(x) { }
26 else { }
27
28 // if statements with exactly one braced statement
29 if(x) {
Line 1, Column 1

Then, we load SeqAn's *uncrustify.cfg* which is located in *${CHECKOUT}/misc*. We can do so by selecting *Indenter > Load Indenter Config File* in the program menu.



Then, we load a file from the SeqAn repository, for example *apps/sak/sak.cpp*.

Now, we can toy around with the reformatter by checking *Live Indent Preview*.

The settings on the left panel allow us to tweak the style to our liking. Any changes can be stored by selecting *Indenter > Load Indenter Config File* in the program menu. The source can also be stored, using *File > Save Source File* and *File > Save Source File As...*.

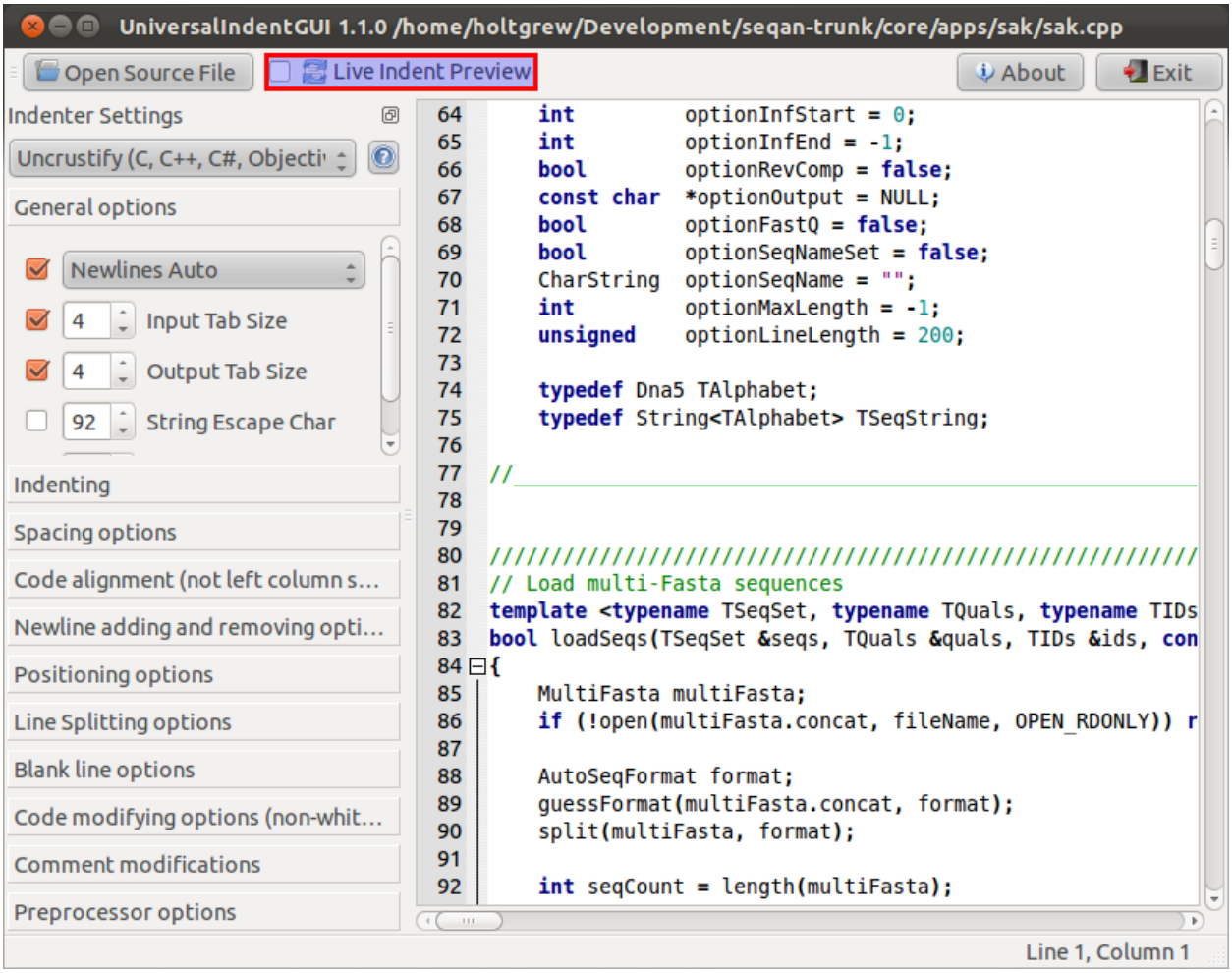
Using The Command Line

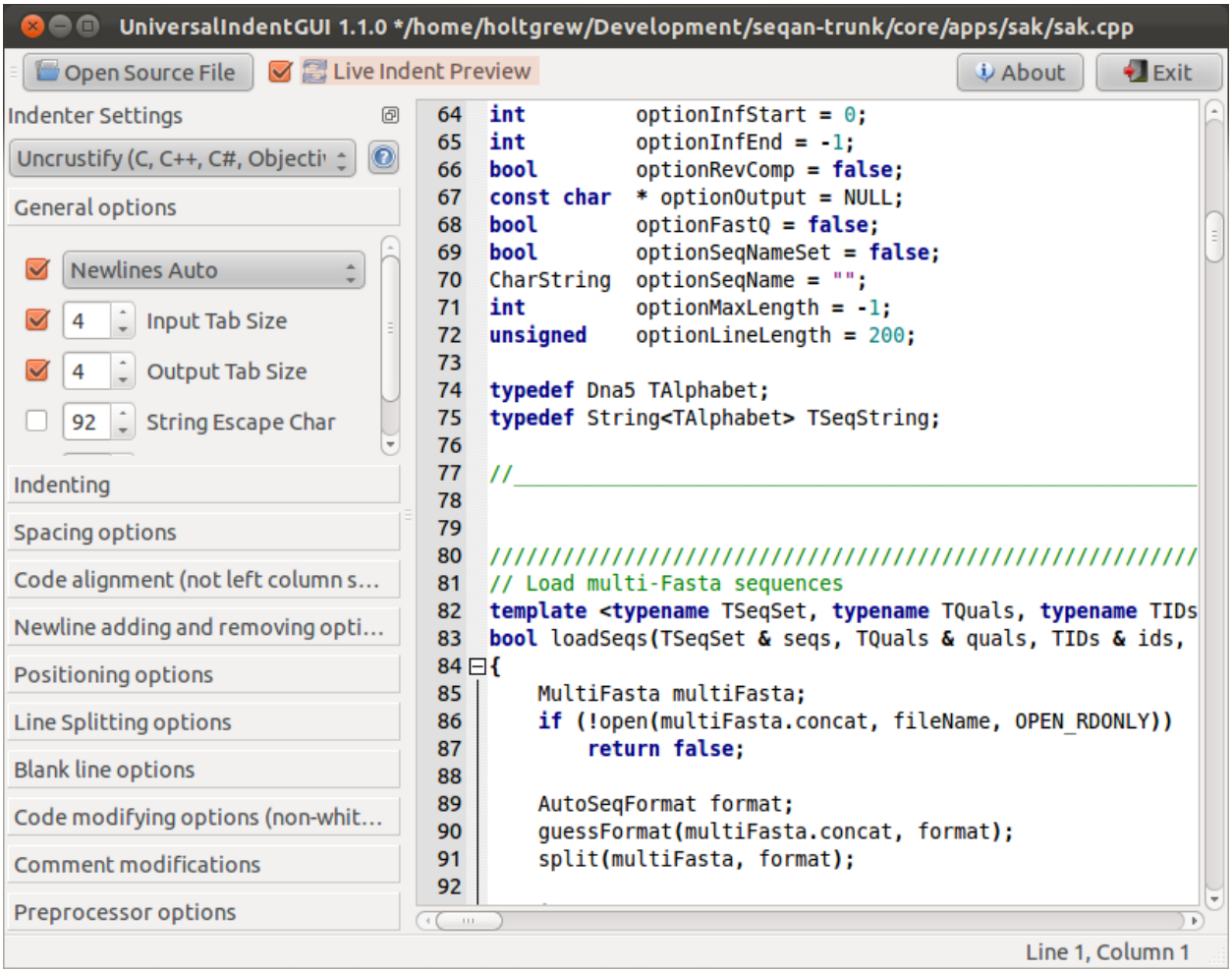
Uncrustify can also be used via the command line. This is best done after a rough visual verification that the uncrustify.cfg yields works for your sources using the Universal Indenter UI.

Work on a single file:

```
# uncrustify -c ${CHECKOUT}/misc/uncrustify.cfg --replace -f path/to/file.cpp
```

Batch work:

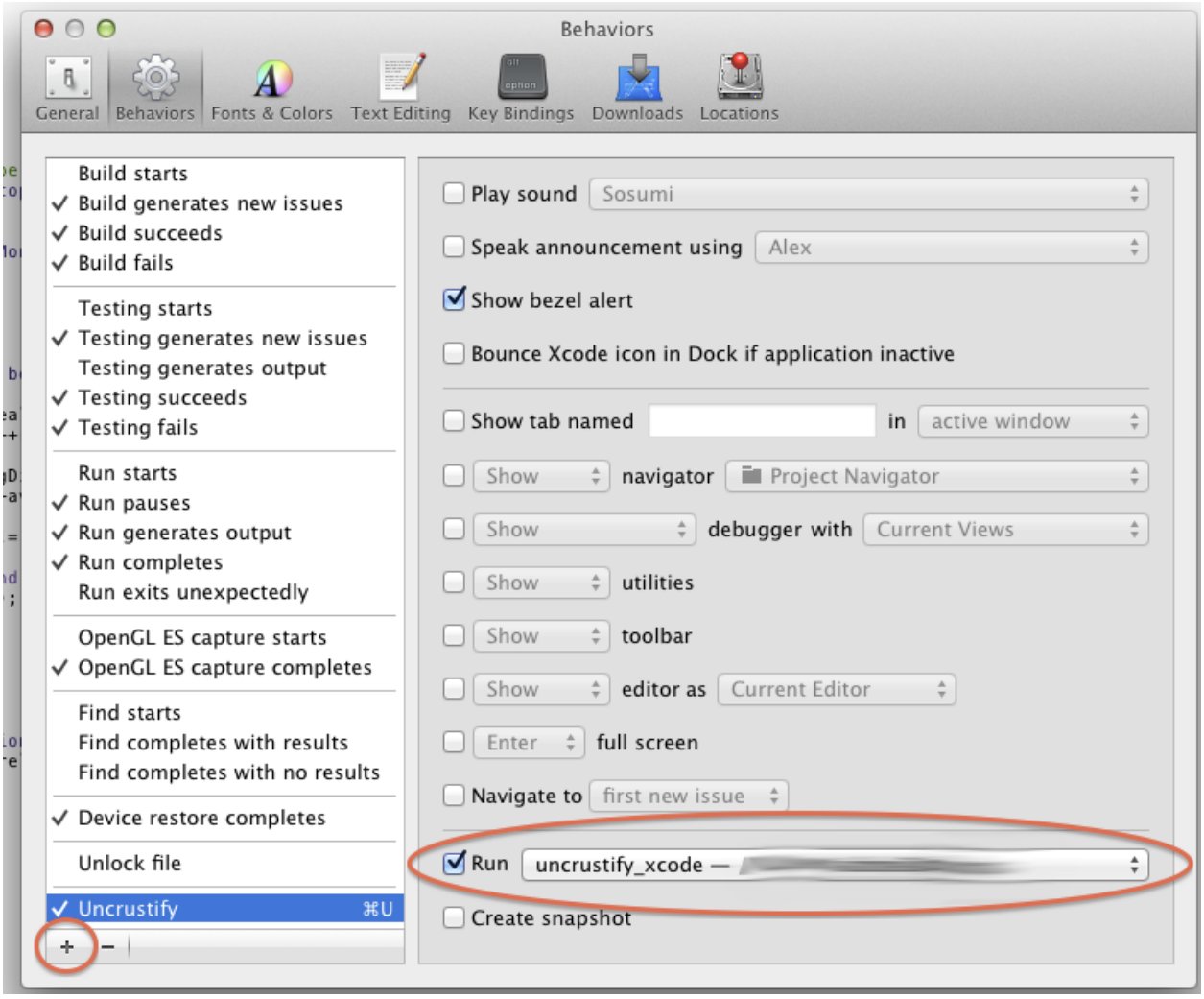

UniversalIndentGUI 1.1.0 /home/holtgrew/Development/seqan-trunk/core/apps/sak/sak.cpp
Open Source File
Live Indent Preview
About
Exit
Indenter Settings
Uncrustify (C, C++, C#, Objecti
General options
Newlines Auto
4
Input Tab Size
4
Output Tab Size
92
String Escape Char
Indenting
Spacing options
Code alignment (not left column s...
Newline adding and removing opti...
Positioning options
Line Splitting options
Blank line options
Code modifying options (non-whit...
Comment modifications
Preprocessor options
int optionInfStart = 0;
int optionInfEnd = -1;
bool optionRevComp = false;
const char *optionOutput = NULL;
bool optionFastQ = false;
bool optionSeqNameSet = false;
CharString optionSeqName = "";
int optionMaxLength = -1;
unsigned optionLineLength = 200;
typedef Dna5 TAlphabet;
typedef String<TAlphabet> TSeqString;
// Load multi-Fasta sequences
template <typename TSeqSet, typename TQuals, typename TIDs
bool loadSeqs(TSeqSet &seqs, TQuals &quals, TIDs &ids, con
{
MultiFasta multiFasta;
if (!open(multiFasta.concat, fileName, OPEN_RDONLY)) r
AutoSeqFormat format;
guessFormat(multiFasta.concat, format);
split(multiFasta, format);
int seqCount = length(multiFasta);
Line 1, Column 1


UniversalIndentGUI 1.1.0 */home/holtgrew/Development/seqan-trunk/core/apps/sak/sak.cpp
Open Source File
Live Indent Preview
About
Exit
Indenter Settings
Uncrustify (C, C++, C#, Objecti
General options
Newlines Auto
4
Input Tab Size
4
Output Tab Size
92
String Escape Char
Indenting
Spacing options
Code alignment (not left column s...
Newline adding and removing opti...
Positioning options
Line Splitting options
Blank line options
Code modifying options (non-whit...
Comment modifications
Preprocessor options
64 int optionInfStart = 0;
65 int optionInfEnd = -1;
66 bool optionRevComp = false;
67 const char * optionOutput = NULL;
68 bool optionFastQ = false;
69 bool optionSeqNameSet = false;
70 CharString optionSeqName = "";
71 int optionMaxLength = -1;
72 unsigned optionLineLength = 200;
73
74 typedef Dna5 TAlphabet;
75 typedef String<TAlphabet> TSeqString;
76
77 //
78
79
80 //
81 // Load multi-Fasta sequences
82 template <typename TSeqSet, typename TQuals, typename TIDs
83 bool loadSeqs(TSeqSet & seqs, TQuals & quals, TIDs & ids,
84 {
85 MultiFasta multiFasta;
86 if (!open(multiFasta.concat, fileName, OPEN_RDONLY))
87 return false;
88
89 AutoSeqFormat format;
90 guessFormat(multiFasta.concat, format);
91 split(multiFasta, format);
92
Line 1, Column 1

```
# find path/to -name '*.cpp' > list.txt
# uncrustify -c ${CHECKOUT}/misc/uncrustify.cfg --replace -F list.txt
```

Automatically fix whitespaces in Xcode

Uncrustify can also be used directly from Xcode. With Xcode 4 Apple introduced so called "Behaviors" that can be executed using for instance keyboard shortcuts. To use Uncrustify you can add a new behavior in the Xcode Preferences (tab Behaviors) and select "*Run*". Here you add the attached ruby script.



Note: The script does **not** uncrustify the currently opened source file but all source files that were changed in your current branch. Xcode does not provide the information which source file is currently opened.

ToC

Contents

Profiling Programs

Linux Perf Tools (Linux)

- https://perf.wiki.kernel.org/
- Requires `echo '-1' > /proc/sys/kernel/perf_event_paranoid` as root.

Useful commands:

- `perf top` - display `top`-like display but on function granularity
- `perf record PROGRAM` - execute PROGRAM with profiling
- `perf report PROGRAM` - display report for PROGRAM

Google Perftools (Linux, Mac Os X)

- Download and install http://code.google.com/p/gperftools/ (also available through Ubuntu/Debian packages)
- Compile your program with debug symbols (you probably want to enable optimization as well).

```
# Tell the profiler where to write its output.
export CPUPROFILE=${OUT}
LD_PRELOAD="/usr/lib/libprofiler.so.0" ${PROGRAM} ${COMMAND_LINE}
google-pprof ${PROGRAM} ${OUT}
```

Interesting commands:

- `gv/web` - display weighted call graph in gv or in your browser
- `top/topX` - display top 10/X hitters
- `disasm NAME` - disassemble functions matching NAME

Instruments (Mac Os X)

Todo

Write me!

ToC

Contents

Writing Nice Unix Programs

In bioinformatics, many programs are of "academic" quality, i.e. they are written to present a new method. The implementation is often "only" used by other academics who, since they do not pay for the program, are willing to take some glitches for granted.

This page tries to help academic software developers in writing better Unix programs. The points raised here come from our own experience with using academic software. The focus is on C and C++ programming in a Unix (e.g. Linux, Mac Os X) environment. The hints should also be applicable to other languages such as Java, and in some way also Windows.

Program Return Codes

Rationale

The `main()` method of a program should be `0` if there were no errors and a value different from `0` otherwise.

Explanation & Reasoning

The `main()` function should return an integer indicating whether the program completed running successfully or not. A value of `0` indicates that no error occurred, a value not equal to `0` indicates that an error occurred. You might consider returning different values for different kinds of errors, e.g. `2` for I/O errors, `3` for logic errors and `1` as a catch-all for any all errors.

This makes it easy for a calling script/program to check whether the program executed successfully.

Example

The following program returns `1` if it receives any argument and `0` if this is not the case.

```
#include <cstdio>

int main(int argc, char ** argv)
{
  if (argc > 1) {
    fprintf(stderr, "I do not like arguments!\n");
    return 1;
  }

  return 0;  // Everything went smoothly!
}
```

The following bash script calls programs and reacts to the return code.

```
#!/bin/sh

# 1. Only success case.
program arg1 arg2 && echo "success!"

# 2. Only failure case.
{|
! echo "failure"
|}


# 3. Handle success/failure case
program arg1 arg2
if [ "$?" ]; then
  echo "success"
else
  echo "failure"
fi

# 4. Use case for separating cases
# TODO
```

Links

- Error Level @ Wikipedia

Assume Few Things About Paths

Rationale

Do not assume anything on paths for (1) the program to reside in (2) temporary files or (3) the working directory. Fixing the program install directory at configure time is OK.

Explanation

Most Unix programs are configured with a `$PREFIX` variable (e.g. setting `--prefix=` in the `./configure` script) and assume that all paths are relative to the given one. For example, the Apache 2 web server reads its configuration from the director`${PREFIX}/apache2`. This is a reasonable assumption. Another reasonable assumption is that the current working directory is writeable. However, temporary files should be stored in `${TMPDIR}` or `/tmp` (see the related section).

Non-reasonable assumptions are:

- *The program is executed in the directory the binary resides in.* For example, program `prog` at path `/path/to/prog` should not assume that the working directory is `/path/to` when it is executed. Especially, do not assume that the directory the binary resides in is writeable. If your program is installed in `/usr/bin`, this path is non-writeable for normal users on Unix.
- A program *must* be in a given specific path fixed at *code writing time*. While it is reasonable for the user to give an install path at *configure-time*, the user should be able to install the program in any directory, including `/opt`, his `${HOME}` directory or `/some-weird-path/the/sys/admin/gave`.

Best practice is:

- Use `${TMPDIR}` if available, fall back to `/tmp`, for intermediate/temporary files.
- Use reasonable defaults for result files, e.g. the path the input file resides in.
- Allow the user to set an output directory.
- Consider asking the user before overwriting result files when using defaults.

Example

Some programs create the result files in the current working directory. This is not good practice, since the current working directory is *context* dependent. While it is possible to use `pushd` and `popd` to use one directory per call to the program, it is much less error prone and more comfortable for the caller to specify the file on the comman dline.

Provide Good Defaults

Rationale

Require as few parameters as possible, provide defaults or guess as many as possible.

Explanation

The more parameters are required in a program, the hard it gets to use. For many parameters, default values can be given by the program's author. Other parameters can be guessed depending on the input.

It should still be possible to override such value by command line parameters.

Example

The quality type of a FASTQ file can be guessed from the file contents very reliably by looking at the quality entries. Nevertheless, the user should be able to override this by a command line parameter.

Positional vs. Named Arguments

TODO

Provide all-in-one-go Variants of your program

Rationale

While many program's steps might add to flexibility, a tool is easier to use if there is only one call to it.

Explanation

Some bioinformatics programs consist of many steps, e.g. (1) building an index (e.g. k-mer or suffix array) (2) perform a search, and (3) combine multiple search results to one. While this might enable the flexible usage of the program it complicates its usage. Please also provide a way to call your program that creates an output from the input files with one program call.

Example

For paired-end read mapping, the program *bwa* consists of multiple calls.

1. Call bwa to build an index on your genome.
2. Map the left-end reads, yielding a position file.
3. Map the right-end reads, yielding a position file.
4. Combine the two position files previously created.

While it is OK to first create an index file (this file can be used for many reads files), the last three steps could be combine into one umbrella command. This would reduce the number of intermediate files and be much more comfortable for users.

Use `stdout` and `stderr` correctly

Rationale

The standard stream `stdout` is for the program's output while `stderr` is for logging and error messages. It should be possible to redired `stdout` to an output file and `stderr` to a log file. Use `-` as shortcuts for `stdout` and `stderr`.

Explanation

In C/Unix programming `stdout` is for output to the user, `stderr` is for error messages and logging. For example, when running daemons (e.g. web servers), the output to `stderr` ends up in log files.

If your program has only one input and one output file, it could accept the input from `stdin` by default and write to `stderr`. An example is the `grep` tool on Unix. You can specify different programs on the command line, however.

If you have program arguments for input and output files then you should use `-` for shortcuts to `stdint` and `stderr`. For example, a call to `program --in-file - --out-file -` would read from `stdin` and write to `stdout`.

Example

- When the program is called with wrong parameters, the return code should not be `0` and the help should be printed to `stderr`.
- When the program is called with a `--help` parameter, the return code should return `0` and the help should be printed to `stdout`.

Allow specifying all file names through the command line

TODO

Do Not Require A Specific Working Directory

Rationale

Do not require that the current working directory is in any relation to the directory containing the binary.

Explanation

Some programs must be called with `./program`, e.g. the current working directory. This makes it harder to use the program when installed centrally and when multiple instances are called at the same time on the same file system. This makes it harder to use in complex software pipelines. Here, additional working directories and either symbolic links or copies of the program binary have to be created for each called instance.

Use `$TMPDIR` For Temporary Files, Fall Back to */tmp*

Rationale

Use the value of the environment variable `${TMPDIR}` for temporary files. If it is not set, use `/tmp` or `/var/tmp`.

Explanation

On Unix, the canonical place to store temporary file is the value of the environment variable `${TMPDIR}`. If it is not set, then use `/tmp` or `/var/tmp`. `/tmp` might be cleared during system reboots while `/var/tmp` is not cleared during system reboots but possibly rather depending on the file age.

Links

- TMPDIR @ Wikipedia

Misc Links

- Heng Li's "Debugging Memory Problems" (Heng Li of BWA, samtools etc. fame)

References

Bibliography

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

[AC75] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

[AGM+90] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990. doi:10.1016/S0022-2836(05)80360-2.

[AM97] Eric L Anson and Eugene W Myers. Realigner: a program for refining dna sequence multi-alignments. *Journal of Computational Biology*, 4(3):369–383, 1997. doi:10.1145/267521.267524.

[BYN99] Ricardo Baeza-Yates and Gonzalo Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999. doi:10.1007/PL00009253.

[BCGottgens+03] Michael Brudno, Michael Chapman, Berthold Göttgens, Serafim Batzoglou, and Burkhard Morgenstern. Fast and sensitive multiple alignment of large genomic sequences. *BMC bioinformatics*, 4(1):66, 2003. doi:10.1186/1471-2105-4-66.

[BDC+03] Michael Brudno, Chuong B Do, Gregory M Cooper, Michael F Kim, Eugene Davydov, Eric D Green, Arend Sidow, Serafim Batzoglou, NISC Comparative Sequencing Program, and others. Lagan and multi-lagan: efficient tools for large-scale multiple alignment of genomic dna. *Genome research*, 13(4):721–731, 2003. doi:10.1101/gr.926603.

[Car06] Reed A Cartwright. Logarithmic gap costs decrease alignment accuracy. *BMC bioinformatics*, 7(1):527, 2006.

[FM01] Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1):13–28, 2001. doi:10.1016/S0020-0255(01)00098-6.

[GKS03] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. *Software: Practice and Experience*, 33(11):1035–1049, 2003. doi:10.1002/spe.535.

[Got82] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

[Gus97] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997. ISBN 0-521-58519-8.

[Hor80] R Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980. doi:10.1002/spe.4380100608.

[LTP+09] Ben Langmead, Cole Trapnell, Mihai Pop, Steven L Salzberg, and others. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol*, 10(3):R25, 2009. doi:10.1186/gb-2009-10-3-r25.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MWM+08] Ali Mortazavi, Brian A Williams, Kenneth McCue, Lorian Schaeffer, and Barbara Wold. Mapping and quantifying mammalian transcriptomes by rna-seq. *Nature methods*, 5(7):621–628, 2008. doi:10.1038/nmeth.1226.

[Mye99] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999. doi:10.1145/316542.316550.

[Pea90] William R Pearson. Rapid and sensitive sequence comparison with fastp and fasta. *Methods in enzymology*, 183:63–98, 1990. doi:10.1016/0076-6879(90)83007-V.

[REW+08] Tobias Rausch, Anne-Katrin Emde, David Weese, Andreas Döring, Cedric Notredame, and Knut Reinert. Segment-based multiple sequence alignment. *Bioinformatics*, 24(16):i187–i192, 2008. doi:10.1093/bioinformatics/btn281.

[RKD+09] Tobias Rausch, Sergey Koren, Gennady Denisov, David Weese, Anne-Katrin Emde, Andreas Döring, and Knut Reinert. A consistency-based consensus algorithm for de novo and reference-guided sequence assembly of short reads. *Bioinformatics*, 25(9):1118–1124, 2009. doi:10.1093/bioinformatics/btp131.

[THG94] Julie D Thompson, Desmond G Higgins, and Toby J Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22(22):4673–4680, 1994. doi:10.1093/nar/22.22.4673.

[Ukk85] Esko Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1):132–137, 1985. doi:10.1016/0196-6774(85)90023-9.

[UPA+14] Gianvito Urgese, Giulia Paciello, Andrea Acquaviva, Elisa Ficarra, Mariagrazia Graziano, and Maurizio Zamboni. Dynamic gap selector: a smith waterman sequence alignment algorithm with affine gap model optimisation. In *In Proc. 2nd Int. Work-Conf. Bioinform. Biomed. Eng.(IWBBIO): 7-9 April 2014; Granada*, 1347–1358. Copicentro Granada SL, 2014. URL: http://iwbbio.ugr.es/2014/papers/IWBBIO_2014_paper_143.pdf.

[WS08] David Weese and Marcel H Schulz. Efficient string mining under constraints via the deferred frequency index. In *Advances in Data Mining. Medical Applications, E-Commerce, Marketing, and Theoretical Aspects*, pages 374–388. Springer, 2008. doi:10.1007/978-3-540-70720-2_29.