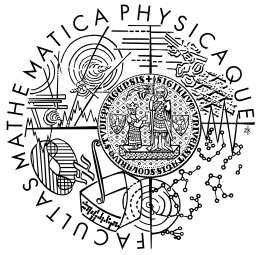

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jaroslav Jindrák

C++ Runtime for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Mgr. Martin Děcký, Ph.D.
Study program:	Computer Science
Study branch:	Software Systems

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: C++ Runtime for HelenOS

Author: Jaroslav Jindrák

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký, Ph.D., Department of Distributed and Dependable Systems

Abstract: In order for an operating system to support running programs written in any given programming language, it needs to provide a runtime environment for that language. The structure of the runtime varies depending on the language, but it generally consists of a tool allowing the program to run, which can be an interpreter or a runtime library, and a standard library providing functions and types used by the program.

In this thesis we evaluate which parts of the C++ runtime are the most needed to support existing programs written in C++ and implement them for the HelenOS operating system. We then port an already existing open-source program written in C++ to verify the correctness of both our research and our implementation.

Keywords: C++, runtime, HelenOS, microkernel

Contents

1. Introduction

In order to run programs written in a given programming language, an operating system needs to have a runtime for that language. In most cases, this runtime consists of a standard library (which contains a set of basic utilities and data containers). Additionally, a runtime of a language can contain an interpreter or, as is the case with C++ , a runtime library. A C++ runtime library contains functions used for tasks such as Runtime Type Identification, static variable lifetime management and stack unwinding.

1.1 HelenOS

HelenOS [1] is a portable microkernel-based operating system designed and implemented from scratch. It was originally created as a software project at the Faculty of Mathematics and Physics at the Charles University in 2006 and has since then hosted multiple student theses and software projects.

Currently, HelenOS contains native implementation of only one runtime - a non-standard runtime for the C programming language[1]. This means that if one wants to implement a program native to HelenOS that can be added to the main repository, they have to write that program in C. However, C does offer programmers very little when it comes to abstraction and safety and thus is not as well suited for user space application development as more high level languages, such as C++ , might be.

1.2 Goals

When deciding on the topic of this thesis, we faced the decision whether we would port an existing implementation of the C++ runtime or implement one from scratch. If we were to port an existing implementation, we would gain a complete, optimized and tested solution. However, in the spirit of HelenOS, we decided to write one from scratch because of the following reasons:

- A ported runtime would not be able to be merged into the mainline HelenOS repository which would lead to the impossibility of using C++ to implement native programs distributed with HelenOS (including user space applications, drivers and servers[2]). This on itself would not be a big problem, but some HelenOS developers voiced their desire to be able to implement servers in C++ and a native implementation would allow for that.

- HelenOS builds its own cross-compiler, which has its version hard coded in the build system. This means that the version of the compiler will only increase in the future and as such our implementation would be able to utilize newer features and techniques (such as variadic templates or expression

[1] Some other runtimes, such as the Python one, are available through ports.

[2] Daemon-like programs that mediate communication between drivers and the kernel.

SFINAE[3]) to achieve a much more readable and elegant source code of the implementation.

- As we mentioned previously, the C library present in HelenOS is non-standard, which means that some of its features are either missing, renamed or have different semantics. Since the C++ standard library is built with the use of the C library, this would require possibly large scale modifications to any library we would be porting.

Implementation of the entire runtime would exceed the scope of a master thesis and as such we will create only a subset of it that will suit for implementation and porting of commonly used C++ features. Note, however, that the implemented subset shall (unlike the C runtime) be in compliance with the official C++ standard. The scope of the implementation will be discussed in Section 2.

The standard version we were following in our implementation is the C++ 14 draft N4296 [2][4] (the standard). However, the C++ 17 standard was released during the work on this project and we took the liberty of using some of its features to implement the standard library to obtain more readable code.

1.3 Thesis Structure

In this first section, we have introduced the topic of this thesis and provided our reasoning for its choice. The following section, Section 2, contains a further explanation of what the C++ runtime consists of and researches which components of the runtime should be part of our implementation. Section 3 then explains some advanced features and techniques of the C++ programming language used in our implementation as well as documentation for the various modules of the implemented standard library. Afterwards, in Section 4, we show our readers how a C++ program could be written for or ported to HelenOS. To demonstrate the results of our implementation, we will present a project written using C++ in Section 5 and, lastly, we will summarize our work and discuss possible future steps.

[3] A newer C++ feature explained in a later section of this thesis, commonly used in template metaprogramming.

[4] This is the C++ 14 standard with some editorial fixes.

2. Analysis

In this section, we will explore the contents of the C++ runtime and make a decision as to which parts of the runtime we will implement in this thesis. As we have previously mentioned, the C++ runtime consists of two parts – a runtime library and a standard library. Each of these libraries will be discussed in one of the following two sections.

2.1 Runtime Library

A C++ runtime library is a set of functions and types that are used by the compiler to perform runtime tasks (such as dynamic casting or calling constructors of static global variables). These functions can either be linked to a program (in which case calls to them would be inserted into the executable by the compiler) or be a part of the environment (e. g. can be called by the program loader).

Since the compiler links this library to a program, any implementation of the runtime library has to conform to the Application Binary Interface (ABI) used by the compiler. Both of the compilers used by HelenOS, GCC [3] and Clang [4], use the same ABI for any C++ programs they compile – the C++ Itanium ABI [5] (the ABI). Despite its name[1], this ABI is used by many major compilers on all major architectures.

The ABI specifies, among others, the way data are laid out in memory, the virtual table structure, calling conventions, name mangling, Runtime Type Identification, static variable lifetime management and exception support. The latter three are the primary parts of the runtime we will be implementing and will be discussed in greater detail in the following four sections.

2.1.1 Runtime Type Identification

Runtime Type Identification (RTTI) is a feature of the C++ language that provides its users with information about the type of an object at runtime. The standard defines three members of RTTI – the class `type_info`, the `typeid` operator and the `dynamic_cast` operator.

typeid and type_info

The `typeid` operator returns a reference to an object of type `type_info`. The `type_info` class is defined by the standard and is used in three situations:

1. To support the `typeid` operator.
2. To match an exception handler with a thrown object.
3. To implement the `dynamic_cast` operator.

[1] The name is a historical artifact, this ABI was initially developed for the Itanium architecture.

While the standard does not require the `typeid` to return references to the same `type_info` instance for multiple type checks of the same type, the ABI does require it to do so. Because of this, the compiler creates instances for each type that requires it (because of one of the three cases mentioned above).

Since different types might need to store different information in their respective `type_info` instance (e. g. list of base classes for the use with the `dynamic_cast` operator), the ABI defines several types that derive from `type_info` and are used by the compiler. These types are:

- `__fundamental_type_info`, used for primitive types
- `__array_type_info`, used for arrays
- `__function_type_info`, used for functions
- `__enum_type_info`, used for enums
- `__class_type_info`, used for classes that have no bases and as a base class for the other class type info types
- `__si_class_type_info`, used for classes with a single public non-virtual base
- `__base_class_type_info`, internal type info for representing bases
- `__vmi_class_type_info`, used for classes that cannot be represented by `__si_class_type_info` (e. g. because of multiple, non-public or virtual inheritance)
- `__pbase_type_info`, used as a base for the two following pointer related structures, contains information about the const volatile (and other) qualifiers of the pointee and a pointer to the type info representing the type of the pointee
- `__pointer_to_member_type_info`, used for pointers to members (i. e. objects that are contained in a type such as a class or a struct)
- `__pointer_type_info`, used for all other pointers

In order for a runtime to support the `typeid` operator, all it has to do is implement the `type_info` class and the derived classes listed above.

dynamic_cast

The `dynamic_cast` operator is used to safely convert pointers and references up, down, and sideways along the inheritance hierarchy. If such a conversion is not possible, the invocation of the operator either returns `nullptr` (when dealing with pointers) or throws an exception (when dealing with references). Since the actual type of its argument is (mostly) known only during runtime, the `dynamic_cast` operator must be (at least partially) implemented as part of the runtime library.

Invocations of `dynamic_cast` that convert a pointer to a possibly cv-qualified[2] void pointer, to a null pointer or invocations that are static are inserted inline by

[2] A type is cv-qualified if it is either const, volatile, or both.

the compiler. This leaves only those cases that are truly dynamic to the runtime library, where they are handled by the function `__dynamic_cast`, whose signature is shown in Listing 1.

```
extern "C"
void* __dynamic_cast(const void* sub,
                     const __class_type_info* src,
                     const __class_type_info* dst,
                     ptrdiff_t src2dst_offset);
```

Listing 1: Signature of the function that performs the dynamic cases of dynamic_cast.

These cases are [2, Section 5.2.7 (8)]:

- Downcasting from a public base class to a derived class (as can be seen in Listing 2).
- Sidecasting between two public base classes of a derived class (as can be seen in Listing 3).

```
class A { /* ... */ };
class B: public A { /* ... */};

A* a = new B{};
B* b = dynamic_cast<B*>(a);
```

Listing 2: Example of a downcast.

```
class A { /* ... */ };
class B { /* ... */ };
class C: public A, public B { /* ... */ };

A* a = new C{};
B* b = dynamic_cast<B*>(a);
```

Listing 3: Example of a sidecast.

In both of these cases, the `__dynamic_cast` function is invoked and receives a pointer to the argument of `dynamic_cast` as its first argument `sub`. Its second and third arguments are pointers to type infos of the (static) type of the variable passed to `dynamic_cast` and of the target type (the fourth argument is a hint that may be passed by the compiler). It then traverses the inheritance graph from `src` trying to reach `dst`, if a path is found, it returns the adjusted pointer, otherwise it returns `nullptr`.

2.1.2 Static Constructors and Destructors

Another feature of the language that needs to be managed by the runtime is the lifetime management of objects with static storage. An object has static storage if it is declared in namespace scope (including global namespace) or if it is declared with the static keyword (which also implies internal linkage) or the extern keyword (which also implies external linkage).

Such objects are allocated when the program begins and deallocated when the program terminates, but the runtime must ensure correct calls to constructors and destructors of these objects. Note that whenever we refer to initialization in the following two sections we mean dynamic initialization (as opposed to static initialization, which is performed at compile time), unless specified otherwise.

Local Static Variables

Variables that are declared at block scope with the static specifier have static storage but according to [2, Section 6.7 (4)] their initialization is performed the first time the control passes through its declaration (though it can be repeated if the initialization fails because of a thrown exception).

```
extern "C"
int __cxa_guard_acquire(int64_t* guard);

extern "C"
void __cxa_guard_release(int64_t* guard);

extern "C"
void __cxa_guard_abort(int64_t* guard);
```

Listing 4: Local static variable guard management functions.

According to the ABI, the runtime library has to define three functions that will manage a guard object defined by the compiler for each local variable with static storage, the signatures can be seen in Listing 4 (and will be referred to as `acquire`, `release` and `abort`).

For each local variable with static storage, the compiler generates a sequence of instructions similar to such that would result from the pseudo-code shown in Listing 5. Here, the code first checks if the variable has been initialized by calling `acquire`, which returns `1` if the initialization has not been completed for the variable and `0` otherwise. In the case when the initialization has not taken place yet, the variable is initialized and `release` is called, which sets the first byte of the guard object to a non-zero value to mark the guarded object as initialized.

[2, Section 6.7 (4)] requires that should another thread reach the place of declaration of a local static variable while another thread is in the process of initializing said variable, it has to wait. Because of this, any call to `acquire` has to be accompanied by a call to either `release` or `abort`, which unlock any synchronization primitives locked by `acquire` (in the case of `abort`, which is called when the initialization throws an exception, the guard object is not modified and

```
if (__cxa_guard_acquire(&guard_obj))
{
    try
    {
        /* initialize the object */
    }
    catch (...)
    {
        __cxa_guard_abort(&guard_obj);
        throw;
    }
    /* queue object destructor */
    __cxa_guard_release(&guard_obj);
}
```

Listing 5: Example code of what the compiler might generate for local static variable initialization.

the next time control reaches this place it will attempt to initialize the variable again).

In order for local variables with static storage to be destroyed at program termination and in the correct order, the runtime queues their destructor using the `__cxa_atexit` function, which behaves similarly to the `atexit` function from the C standard library. Note that due to this similarity this function will not be discussed in greater detail and should one desire to learn more about its semantics, the place to look for is section 3.3.5.3 of the ABI [5].

Global Static Variables

Unlike for local static variables, the ABI does not specify how a compiler or the runtime should handle the initialization of global variables with static storage. According to [2, Section 3.6.2 (4)], it is implementation defined whether the initialization of a non-local variable with static storage is done before the first statement in the `main` function or whether it is deferred to some point in time after the first statement of `main`[3]

Fortunately, both GCC and Clang store the initialization procedures in the ELF header of the generated binary file. This means that the task of global static variable initialization is not handled by the compiler nor by the runtime library, but by the program loader of the system, which parses the ELF header of the binary file being executed. Currently, the HelenOS program loader does not parse either of the ELF sections used to store initialization routines (called `.init_array` and `.ctors`), however GCC provides symbols that refer to arrays that represent these routines. This means that instead of extending the program loader in HelenOS to parse them from the ELF header, we could simply refer to these arrays from within the startup function of libc (the associated destructor

[3]That is, as long as such initialization happens before the first use of any function or variable defined in the same translation unit as the variable being initialized.

functions are registered similarly to those of static local variables).

2.1.3 Exception Support

The ABI defines two levels of the stack unwinding interface[4]. The first level is a language agnostic unwinding library, which can be used to implement exception support (such as that in C++) or e. g. the `longjmp` function in the C standard library. Functions and types included in this level range from register manipulation to raising exceptions to the actual biphasic unwinding process (stack search and actual unwinding, note that C++ does not require two phases in the unwinding process, but the ABI requires their presence for C++ to be able to coexist with other languages on the stack). The second level describes C++ specific parts of the unwinding process, including data allocation, deallocation, initialization and layout in memory.

Between the two levels the ABI defines personality routines. These are functions that are called for each stack frame during the unwinding process and serve as a mediator between the language agnostic unwinding library and the language specific exception support library defined in the second level.

While it stands to reason we would need to implement the second level as part of our implementation, the first level is not necessarily a part of only the C++ runtime and thus often exists as a separate entity. The version of HelenOS for which we wrote our implementation, however, does not offer any implementation of the level one unwinding interface.

2.1.4 Conclusion

Now that we have examined what it entails to implement the different parts of the C++ runtime, what remains is to decide which of these features we will be writing as part of our implementation:

RTTI

To support RTTI, the implementation needs to contain the definition of several simple structures that act as specialized type infos for different classes of types. Additionally, the implementation of a single function (`__dynamic_cast`) is also needed.

Since the implementation of the type info derivatives is fairly simple and the `dynamic_cast` operator is one of the C++ conversion operators [5] which allows us to convert between polymorphic objects in a safe manner, we decided to include this module in our implementation.

Local Static Variables

While trivial in implementation complexity, this feature enables programmers to make use of the so called Meyers singleton [6]. This technique, an example of which can be seen in Listing 6, represents an easy way to implement instances of the singleton design pattern that have lazy and thread safe initialization. For

[4]It actually defines three levels, but the third level represents suggestions for implementation.

[5]The others being `static_cast`, `reinterpret_cast` and `const_cast`.

this reason alone we find the inclusion of this feature in our runtime library to be worth the effort of implementing it and will be including it in our implementation.

```
class Singleton
{
    // Disable instantiation and copy.
    Singleton() = default;
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    public:
        static Singleton& instance()
        {
            // Initialized when control first reaches this place,
            // lives for the entirety of the program runtime.
            static Singleton inst{};

            return inst;
        }
};
```

Listing 6: A minimalistic example of the Meyers singleton.

Global Static Variables

This feature of the runtime library explicitly requires us to modify existing C code in HelenOS, specifically the program loader and ELF parser. Fortunately, a previous thesis [7] implemented a GCC specific extension for the C programming language that allows program to denote functions as constructors and destructors (which then act in the same way as constructors and destructors of global static variables in C++ do). Both C++ and C constructors are stored in the `.init_array` sections of the ELF header, so we can use this thesis as a referential implementation. However, C++ and C destructors do not function the same according to the ABI, because the C destructors are stored in another ELF section (called `.fini_array`) and C++ destructors are registered to the C++ runtime equivalent of the C standard `atexit` function when the relevant constructor is executed.

This also implies that the support for C++ destructors of global static variables requires C++ specific code inside the program loader of HelenOS. Since the loader will be the same for both C and C++ programs in HelenOS, we would need a way to call functionality of the C++ runtime library in a program loader designed to support only C programs. This can be achieved by using the C standard function `atexit` (by registering the C++ finalization function the first time a destructor is registered with it) which, however, is not part of the standard C library in HelenOS and would thus require implementation.

While the scope of the implementation of this feature is non-trivial and requires us to invade existing HelenOS ecosystem, some objects in the C++ standard library rely on it. This includes the `cin` and `cout` objects used for standard I/O operations which, as we will see in Section 2.2, are commonly used in C++ programs.

Without the implementation of this feature, these basic objects would not function properly (in the least they would not be able to reference each other for the sake of I/O synchronization). Because of this, we think that the implementation of this C++ runtime feature is a necessity and will be including it in our runtime library.

Exception Support

The implementation of stack unwinding support is a much larger endeavor than the implementation of the remaining parts of the runtime library both in complexity (a much lower level approach not unified across architectures[6]) and in size. When we started our work on this project, there were efforts to implement a HelenOS specific compilation target for GCC that would potentially allow us to use the level one unwinding implementation of `libgcc`. This compilation target has been finished [8] during the implementation of our runtime, but only for a newer version of HelenOS and not for the one our project was based on. If we were to use this implementation, we would be able to avoid most, if not all, architecture specific parts of the stack unwinding functionality, but it would require us to rebase our code base and we would still need to implement the C++ specific part of stack unwinding.

Additionally, a recent developer survey [9] indicated that over half of over 3000 developers reported exceptions being either partially (32.10%) or fully (20.03%) disallowed in either work or school projects they were working on. Because of this and the non-triviality of the implementation, we decided to not implement the exception unwinding API in our runtime library as part of the project and leave that for future work.

2.2 Standard Library

The standard library of a programming language provides users of that language with utilities used in software development that need not be a part of the language itself and have been deemed by the designers of the language to be of such use that they should be available to each and every program written in the given language. Unlike the runtime library, which generally does not provide its functionality to the users via a public interface, the standard library of a programming language follows well-defined signatures and semantics of all functions and types it consists of.

Similarly to the runtime library, the implementation of an entire standard compliant library is out of the scope of this thesis (the library is defined in over 700 pages [10] of the standard). Because of this, we will examine the contents of the standard library and decide what subset of it we will be implementing in this thesis in the following sections.

2.2.1 Library Statistic

In order to be able to determine which components of the standard library should be included in our implementation, we need to find a suitable metric that will

[6] For example the ARM architectures require a different unwinder.

allow us to compare them. While we cannot predict which parts of the library will be used in programs that are yet to be written, we can analyze how often they are used in existing programs by scanning existing public code bases. The source of these code bases for us will be GitHub, a web-based code hosting service which hosts almost 57 million repositories and is used by almost 20 million users [11] (and is also the platform that HelenOS itself is hosted on).

For simplicity, we will discuss the merit of the different parts of the library in the scope of the individual headers as defined by the standard. Besides granting simplicity, this approach lets us avoid missing intra-header dependencies which would not be easily spotted if we used a symbol based statistic. However, in addition to intra-header dependencies, we need to take into account inter-header dependencies. These are sometimes explicitly stated in the standard, but in some cases might not be noticeable from reading the standard itself. For example, the <ios> header contains the types `ios_base` and `basic_ios`, which serve as base classes for most of the other types in the I/O library module. Because these types are seldom useable on their own, however, this header is not included by projects as commonly as other I/O headers. To also minimize our chances to miss these inter-header dependencies, we will also compare entire modules of the library (as defined in the standard with some exception as will be noted in Section 2.2.3) and then compare headers within those modules.

The statistic, which will be explored in the following sections, is based on the data from 270 popular repositories[7] written primarily in the C++ language (enough so that the repositories get listed when one filters search results by this language). Every considered header from the standard library was assigned a numeric counter that represents the rate of inclusions in these projects. In each of the searched repositories, we have scanned all files that contain C++ source code (based on common C++ source file extensions [12]). For each of these files, we incremented a header's counter *if* it was included. Note that this means that each project can contribute to any inclusion counter by adding at most one and thus large projects will not overshadow small projects (the analyzed projects range from hundreds to hundreds of thousands of lines of code).

Any and all of the scripts used to generate plots presented in the following sections can be found as attachments of the electronic version of this thesis and will be listed and described in the Attachments section.

2.2.2 Standard C Headers

Before we start investigating the different headers that the standard library contains, we must note that [2, Section 17.2] requires us to include (slightly modified) headers from the standard C library. These headers are available to C++ programmers via the use of special wrapping headers that are prefixed with the letter `c` and do not have an extension (e. g. <cstdio> is the C++ version of the standard C header <stdio.h>).

Note that while the standard requires minor modifications to these headers (such as omitting the `restrict` keyword or explicitly requiring C linkage from the

[7] This (seemingly arbitrary) number is based on the limit for unauthenticated requests to the GitHub API and ensures that any of our readers can get the same amount in one execution of our scripts.

compiler), the actual contents are dependent entirely on the standard C library implementation. As we have mentioned before, this library in HelenOS is not standard compliant, which means that some features of the library are missing, have different signatures or different semantics.

Because of this, we will not be including any of these headers in our research and will implement wrappers for those of these headers that are present in the HelenOS standard C library.

2.2.3 Modules

The standard defines 13 library modules that group individual header files together based on their common purpose and use. As previously mentioned, we decide to examine the standard library at the module level and then compare individual headers inside each module in order to decide which header files we will be providing as part of our implementation. The reason for this was that header files in the same module might use the same utility functions (e. g. two containers, albeit representing data in different ways, will use similar tools such as algorithms, iterator related functions and types and type traits) and use the same base types (e. g. in the case of the I/O module, a special header is provided that contains base classes used by other I/O types that are seldom used for different goals and as such their headers might not be included as often).

An added bonus of this approach is that by assuming relationships between headers within a module, we reduce the complexity of checking for actual dependencies (which can be then done within each module instead between each pair of headers in the library).

The modules, as defined by the standard, are[8]:

- Language support library [2, Section 18] contains most of the C library wrappers as well as C++ specific headers such as generic limits, dynamic allocation operators and basic exception types.
- Diagnostics library [2, Section 19] contains commonly used concrete exception classes and exception wrappers for system specific errors as well as C error handling facilities (<cerrno> and <cassert>).
- General utilities library [2, Section 20] contains, as the name suggests, general utilities for C++ programs such as basic data structures (pairs and tuples), memory management traits and types, smart pointers, function objects and time utilities.
- Strings library [2, Section 21] contains the basic template for strings and wrappers around the null-terminated character sequence utility headers from the C standard library.
- Localization library [2, Section 22] contains facilities for programs to be agnostic of cultural differences, including character classifications, date and time representation, currencies and numeric parsing.

[8]Note that the features listed here are not exhaustive.

- Containers library [2, Section 23] contains basic types used to hold user provided data in both sequential and associative fashion as well as adaptors for these types.
- Iterators library [2, Section 24] contains generic iterators that work on streams or traverse data differently from container iterators, traits and iterator adaptors.
- Algorithms library [2, Section 25] contains library provided algorithms that can be used on (not only) user provided sequences of data.
- Numerics library [2, Section 26] contains numeric algorithms, complex numbers and operations with complex numbers and facilities for pseudorandom number generation.
- Input/Output library [2, Section 27] contains basic stream types that operate on strings and files as well as instances of these streams that manipulate standard input and output provided by the operating system.
- Regular expressions library [2, Section 28] contains types and function used for operations that involve regular expression matching and searching.
- Atomic operations library [2, Section 29] contains objects that provide atomic access to data.
- Thread support library [2, Section 30] contains types and functions used to create and manage threads, perform mutual exclusion and wait for conditions to be met.

In addition to excluding the C library headers from our statistic, we will also be excluding the first module, Language support library. We do this because twelve headers in this module are C library headers, two are part of the runtime library (<typeinfo> and <new>), one provides a generic wrapper over C type limits (<limits>) and is crucial to many templates defined across the library and one is integral for the construction and many member functions of almost all containers in the library (<initializer_list>). Only one header, <exception>, is neither part of the library nor crucial to other modules. However, it is an important part of the Diagnostics library (because it contains its base classes) and thus we decided to move this header to that module for the purposes of our statistic.

Figure 2.1 displays inclusion counts of all headers included in the research. In it, the modules fall into three distinct categories: modules with over 800 inclusions, modules with inclusions between 200 and 400 and modules with less than 200 inclusions. While these numbers may be influenced by the number of headers in each module, this comparison implies possible dependencies and cases of code reusability and lets us examine each of the modules without worrying ourselves with every possible dependency (i. e. providing a heuristic).

Because it is not easy to assess the complexity of implementing each header or module (and explicitly state which headers will get implemented) due to the size of the library as defined by the standard, we will assign priorities to each module determining how important it is for us to be a part of our standard library.

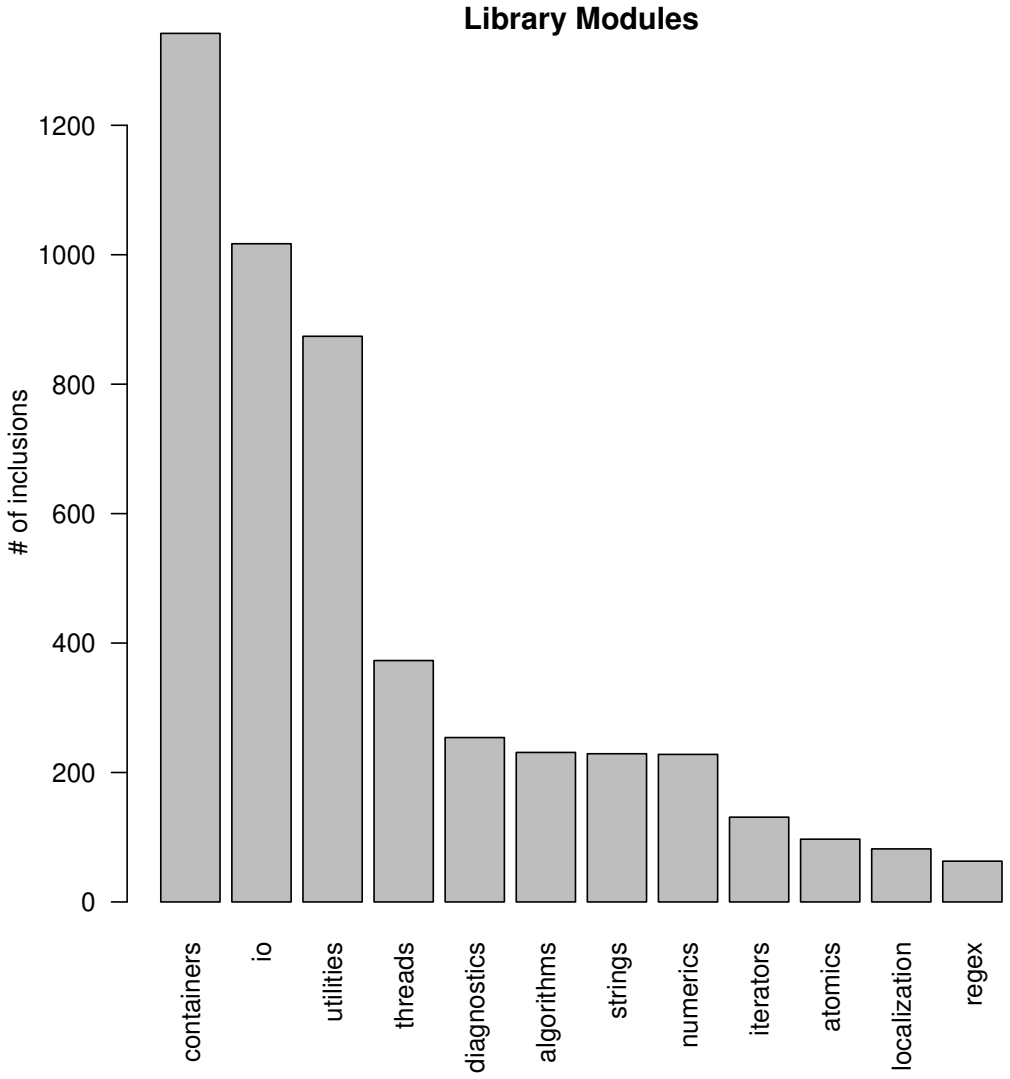

Figure 2.1: Inclusion counts of headers grouped into modules as defined in the standard.

We will do this by using the previously mentioned three categories – this would mean that modules that had over 800 inclusions would have the highest priority, modules that had between 200 and 400 inclusions would have medium priority and modules that had less than 200 inclusions would have the lowest priority.

In the following sections, we will go over each of the modules and the inclusion numbers of their headers, examining the importance of each header within a given module. We will also possibly alter the priorities mentioned in the previous paragraph if we find any important inter-module dependencies.

2.2.4 Containers Library

This module contains data structures and their adaptors that are used to store objects[9] provided by the users of the library. The standard defines three categories

[9]Note that we use the term *object* to refer to an instance of a type, i. e. not an instance of user defined type (e. g. a class) as is common in object oriented languages.

of containers:

- Sequence containers, which organize objects stored in them into a linear arrangement.
- Associative containers, which store pairs of keys and values either ordered or unordered by the keys.
- Container adaptors, which wrap around a different container and provide additional features.

Each of the first three categories provides a unique way of representing data and as such we should implement containers belonging to all three of them in order to satisfy all the needs the users of our library might have. Container adaptors, on the other hand, each cater to a different need. However, these adaptors are simple wrappers around containers from the first three categories and thus their implementation is much simpler.

Sequence Containers

The standard defines the following sequence containers:

- `array`: A fixed sized array that has a size known at compile time.
- `deque`: A double ended queue, allowing insertions and deletions at the beginning and the end.
- `forward_list`: A singly linked list.
- `list`: A doubly linked list.
- `vector`: An array of objects that expands as needed.

One of the things we have to keep in mind while comparing the different symbols defined by the standard is whether we can easily replace them with another symbol should we decide to not implement them. Out of the containers listed above, `array`, `vector` and `deque` are random access containers, i. e. can be index-accessed in constant time. The remaining containers, `list` and `forward_list`, are only forward containers, which means that in order to reach a certain element, we need to iterate over its predecessors in the sequence.

In the set of random access containers, `deque` differs from the remaining containers by not being contiguously allocated in memory (this is implied by the standard requiring references into the container not being invalidated during insertion at either end) which makes it a combination of the `vector` and `list` container. The remaining two random access containers are very similar to each other with `vector` being able to act as `array` while also being able to grow in size as needed. However, the `array` container is a thin wrapper over a C style array and as such is fairly easy to implement.

The `list` and `forward_list` behave similarly to each other, the only major differences between the two being that `forward_list` only keeps pointers to nodes in only one direction (which makes it more lightweight and more suitable for environments with lower amount of memory) and the notion of *before begin iterators*,

which can not be dereferenced and their increment returns iterators that compare equal to iterators returned by the `begin` member function of the `forward_list` instance. Because optimization is not our primary goal, the first difference is unimportant to us. The second difference can be dealt with using either adaptor iterators on `list` iterators or modifying the algorithms in ported code to not use the before begin iterators. This means that the `forward_list` container is not as important because we can substitute its use for `list`.

Associative Containers

Container	Stores	Keys
`map`	key + value	unique
`multimap`	key + value	equivalent
`set`	key	unique
`multiset`	key	equivalent

Figure 2.2: Differences between associative containers.

The standard defines four different associative containers, each in an ordered and an unordered variant, giving us eight in total. The basic differences of these containers (ignoring whether a container is ordered) can be seen in Figure 2.2. Associative containers provide us with the ability to associate a value with a key. This value can be either an actual value (in which case the container stores a *key value pair*) or the existence of such key (in which case the container contains only keys).

The keys in an associative container can be either unique, in which case adding a key that is already present in the container fails, or equivalent, which allows multiple values associated to a single key (or multiple "existences" of a key in containers that do not associate actual values to keys). The containers with equivalent keys behave similarly to those with unique keys. This holds especially in the case of `set` and `multiset`, because their interfaces are nearly identical. While `multimap` could in theory be used in the place of `map`, the latter has member functions unique to it, including a commonly used overloaded `operator[]`. However, the similar interfaces of these four containers potentially allow us to implement them as adaptors over one generic base container, making the implementation easier.

As we previously mentioned, each of these four containers is defined in two variants – one with its elements ordered and one with its elements unordered. While these two variants differ in their interfaces, their basic behavior is very similar and as such one could be used in place of the other. But even so, both of these variants can be implemented using two generic base containers in which case we would get eight containers for the price of two.

Container Adaptors

Container adaptors are used to provide a specific interface using an underlying container for implementation. Because of this, they are just thin wrappers over existing containers. The standard defines three container adaptors in the containers library – `queue`, `stack` and `priority_queue`. The former two can use nearly

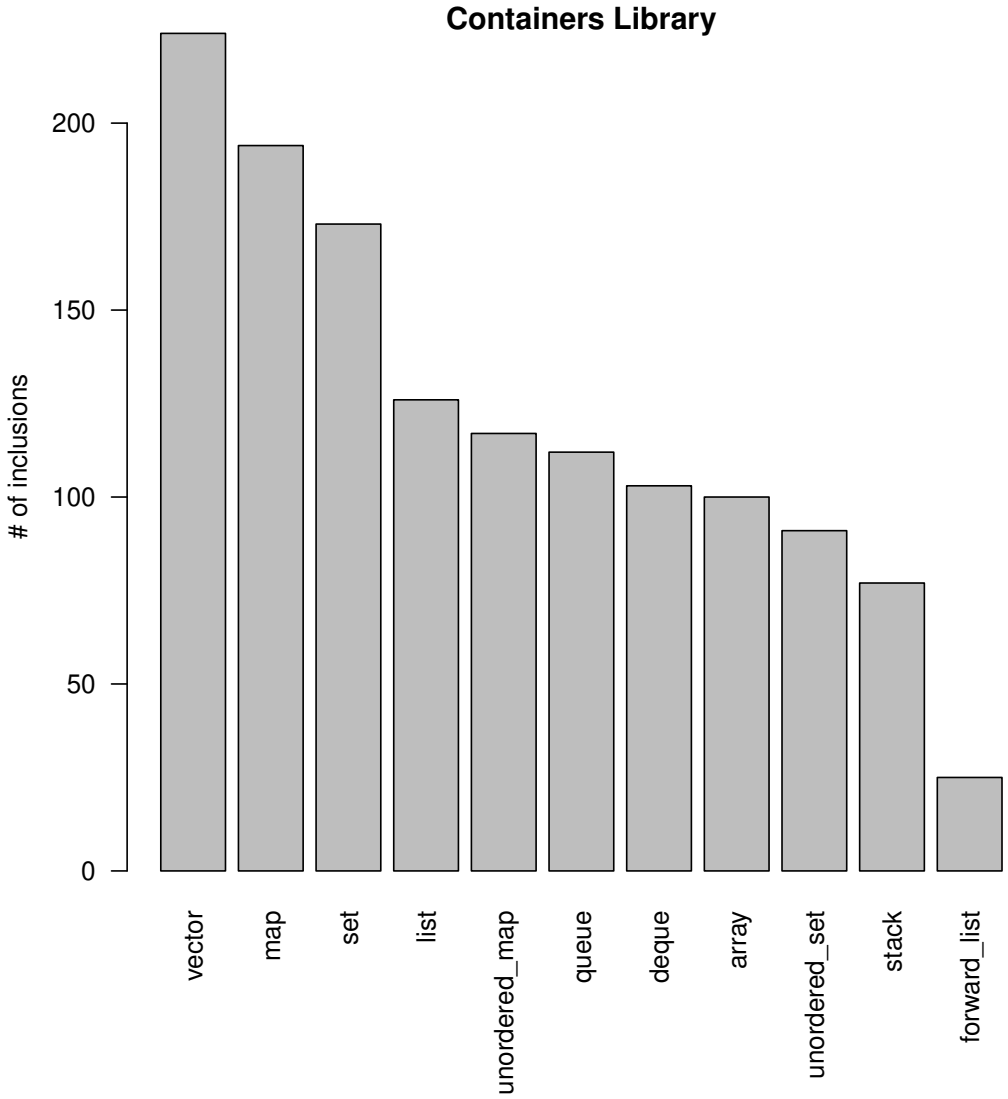

Figure 2.3: Inclusion counts of headers from the containers module.

any sequential container, but the latter requires a heap like behavior (possibly using the heap management functions from the <algorithm> header). Despite this inter-module dependency, all three types are small (when compared to the rest of the module) and would provide our library additional features at low cost.

Conclusion

In our examinations of the individual containers we deemed most of the containers worthy of implementation due to them either being unique in their behavior, easy to implement or able to be implemented as a group. The only container that did not present itself as being important was `forward_list` and the inclusion counts, shown in Figure 2.3, support our decision to assign its implementation a lower priority.

2.2.5 Input/Output Library

The I/O library, as its name implies, provides its users with means to manipulate input and output from and to files (<fstream>), strings (<sstream>) and standard streams (e. g. stdin or stdout via <iostream>). Additionally, this module provides functions and types used to manipulate these I/O facilities as well as means to customize or create new streams (via <streambuf>).

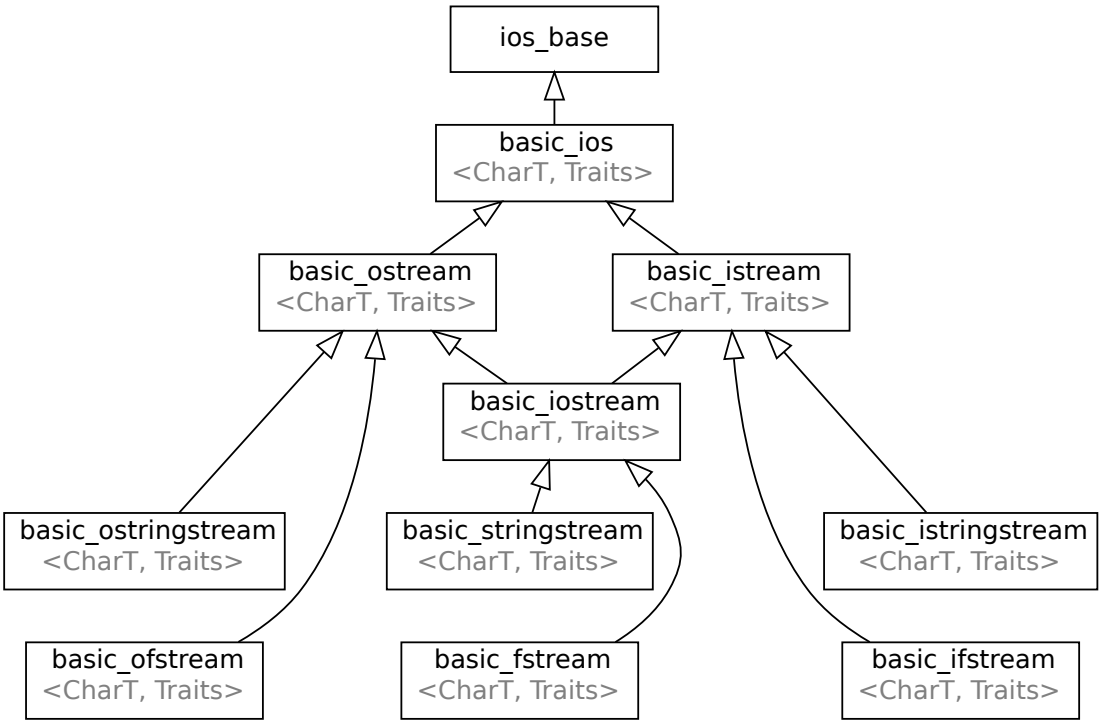

Figure 2.4: Inheritance hierarchy of the I/O streams. (Source: cppreference.com)

This module is the only one in the standard library that relies heavily on inheritance and runtime polymorphism. Highest in the inheritance hierarchy, which can be seen in Figure 2.4, are ios_base (which focuses on flags and state monitoring of the derived I/O types) and basic_ios (which focuses on stream buffers, utility member functions and encoding specific conversions). Both of these types are defined in the header <ios> and are not very usable on their own. This is an example of a header that is not important to the users of the standard library (except for cases where they want to derive from them), but is a crucial intra-module dependency.

On the next level of the inheritance hierarchy, the standard defines two types, basic_ostream and basic_istream, which add stream operators and member functions for input and output manipulation. These types on their own are capable of managing the input and output of a program (the cout and cin objects are instances of these types), but get rarely included because the standard provided stdout and stdin streams are included via the <iostream> header (though they are still usable by programmers that want to inherit from these types).

The remaining types (string based streams and file based streams) utilize the input/output functionality of the iostream types and extend it primarily with custom constructors (used to specify a file to open with opening flags or a string buffer used). As such should be fairly easily implementable in comparison.

Besides this inheritance hierarchy, the behavior of the I/O types can be changed by the usage of different streams. The base class that streams derive from, `basic_streambuf`, is defined in <`streambuf`>. Besides this base class, the standard defines string based and file based derived types (`basic_stringbuf` in <`sstream`> and `basic_filebuf` in <`fstream`>). Additionally, although not defined explicitly, special stream buffers may be required for `stdout`, `stdin` and `stderr`.

Outside of these I/O types, this module contains the header <`iomanip`>, which offers manipulator functions used to manipulate the input or output via overloaded stream operators. Each of the manipulators returns a special function object that changes the state of the I/O type it has been passed to. The standard defines three categories of manipulators:

- Standard manipulators, which change the flags and state (e. g. precision, fill or width) of the I/O types.
- Extended manipulators, which allow for the input and output of locale dependent currencies and times.
- Quoted manipulators, which allow for the input and output of quoted strings.

The latter two categories of manipulators are locale dependent and as such their inclusion depends on our decision to include the localization library, but the standard manipulators offer a wide range of I/O manipulation at the cost of defining few simple function objects (the state and flag manipulation features are already provided by the types `ios_base` and `basic_ios`).

The last header defined in this module is <`iosfwd`>, which serves as a simple forward declaration of each of the I/O types to avoid circular dependency problems in the implementation of the other headers. Because of this, the importance of this header is highly tied to the importance of the other headers and the difficulty of the implementation of this header is miniscule.

In Figure 2.5, we can see the inclusion counts for the aforementioned headers in this module. Most included headers consist of the three that define the commonly used I/O objects (<`iostream`>) and I/O types (<`sstream`> and <`fstream`>). These three headers are tailed by the I/O manipulator functions defined in <`iomanip`>. The remaining headers are all dependencies of the first three and have nearly identical inclusion counts (with the exception of <`ostream`>, which is helpful in the creation of e. g. logging I/O types which may cause it to be included more often).

Because of this, we decided to keep the high priority of these headers given by their existence in the second most commonly used module, with the possible exception of the extended and quoted manipulators from <`iomanip`>, whose priority is bound to the priority of the localization module.

2.2.6 General Utilities Library

Through the general utilities module of the standard library, C++ provides its programmers a wide variety of auxiliary types and functions. Due to the varied

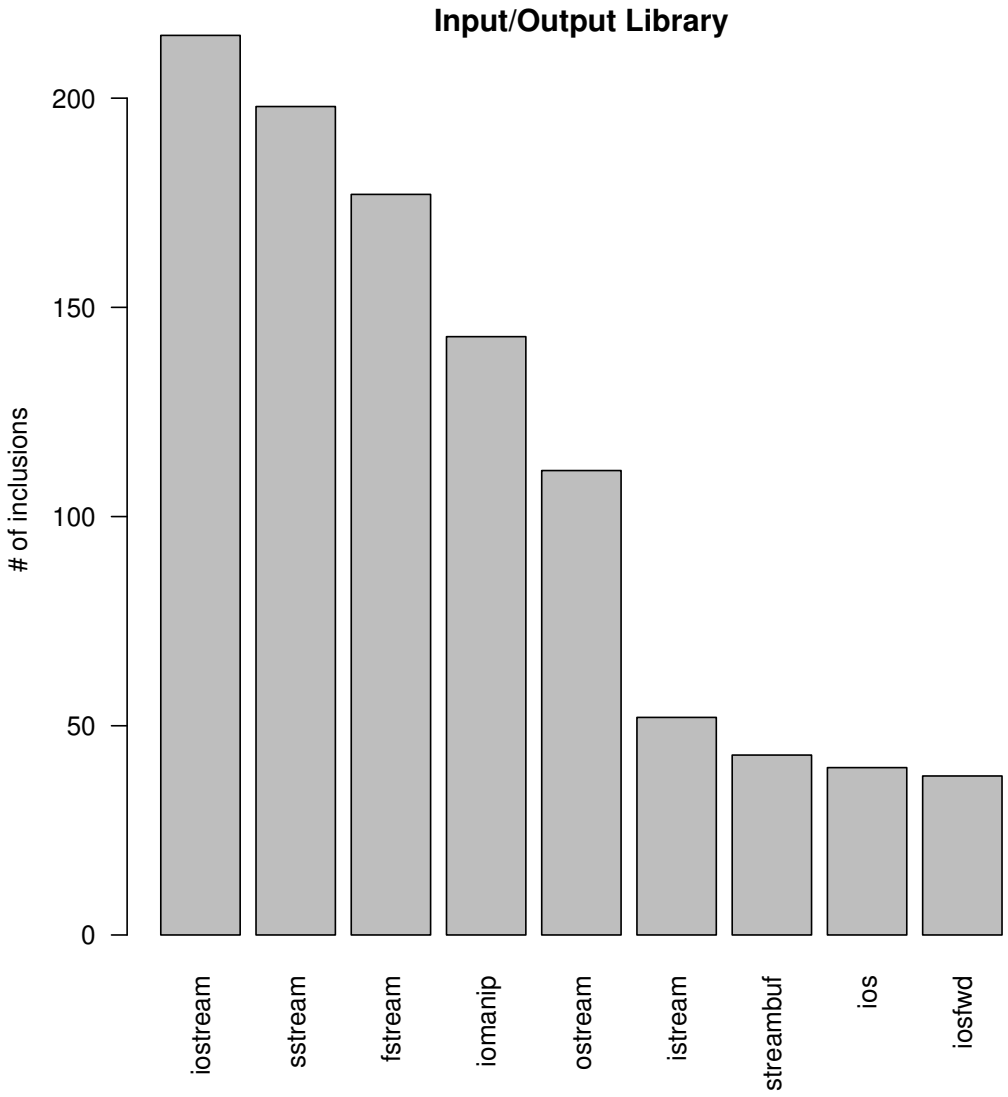

Figure 2.5: Inclusion counts of headers from the I/O module.

nature of the headers defined by this module, we will now examine each individual header in the order as defined by the standard primarily by listing their main contents[10]:

- <`utility`>: class template `pair`, auxiliary functions supporting semantics like moving and forwarding, compile time integer sequences (used to implement e. g. `tuple`)
- <`tuple`>: a compile time heterogeneous container
- <`bitset`>: a bit sequence with size known at compile time
- <`memory`>: allocator, allocator traits, pointer traits, smart pointers
- <`functional`>: reference wrappers, basic arithmetic and logic functors, function object wrappers and bind expressions, class template `hash`

[10]Note that more detailed information will be presented along with developer's documentation in Section 3

- <type_traits>: metafunctions used to test type characteristics and transform types at compile time
- <ratio>: compile time rational arithmetic
- <chrono>: time measuring capabilities (relying heavily on <ratio> in their implementation)
- <scoped_allocator>: an allocator adaptor that can be used with nested containers
- <typeindex>: a type_info wrapper that can be used as key in associative containers

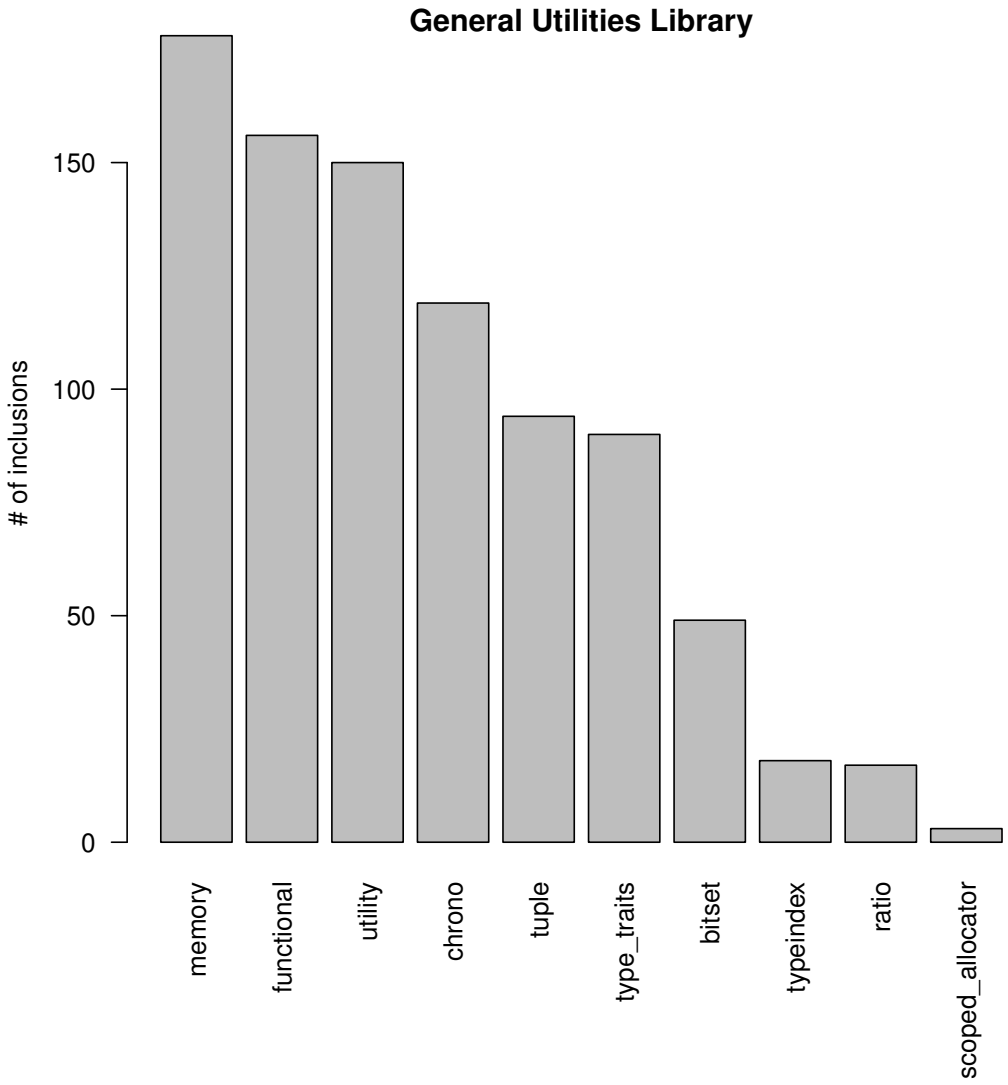

Figure 2.6: Inclusion counts of headers from the utility module.

Figure 2.6 presents us with the inclusion counts of the individual headers from this module. Five of the six most included headers have all great importance to the rest of the standard library (<memory>, <functional> and <utility> primarily for containers, <tuple> for any cases where we need to store a heterogeneous sequence of variables and <type_traits> whenever conditional compilation

is needed). These five and the sixth (<chrono>) due to its inclusion count deserve the high priority of implementation this module's total inclusion counts warrant. The remaining four headers are of more variable use, so we will go over them individually:

- <bitset>: has no use in the rest of the library explicitly stated in the standard and as such we assign it only a medium priority
- <typeindex>: similarly to <bitset> it is not required to implement any other parts of the library and it is included by user code even less than <bitset>, but it is only a simple wrapper over type_info and as such we also assigned it a medium priority
- <ratio>: is a crucial dependency of <chrono> which uses it for time unit representation and conversions, it is also fairly small header and we decided to keep it in the high priority group
- <scoped_allocator>: is not used by any other header in the standard library and has the lowest inclusion count out of all of the standard headers, so we decided to assign it a low priority

2.2.7 Thread Support Library

The standard threading library provides C++ programmers with means to run code concurrently using threads. Unlike most of the other modules, this one tends to be implemented as a wrapper over system specific threading API. This means that in order to be able to implement headers in this module, we need an appropriate libc feature to wrap them around.

The threading support in HelenOS is provided through the fibril API. Besides the actual threading primitive, common synchronization primitives such as mutexes (which can be used to implement <mutex>), read write locks (which, despite the naming difference, function in a similar way as locks found in the standard header <shared_mutex>) and conditional variables (used to implement <condition_variable>).

The only missing feature required by the standard library are promises and futures from the <future> header. Those, however, can be implemented from scratch, although their implementation will then be much harder than the rest of the module.

Given the usability of the features provided by this module and the relative ease of their implementation, this module should be high on our priority list (possibly excluding <future> due to the aforementioned difficulties and leaving it at a medium priority).

2.2.8 Diagnostics Library

The diagnostics library consists of two main part (excluding the C wrapper headers): exception handling support features contained in the header <exception> and exception classes contained in the headers <stdexcept> (generic exception types) and <system_error> (wrappers around system specific error codes and conditions).

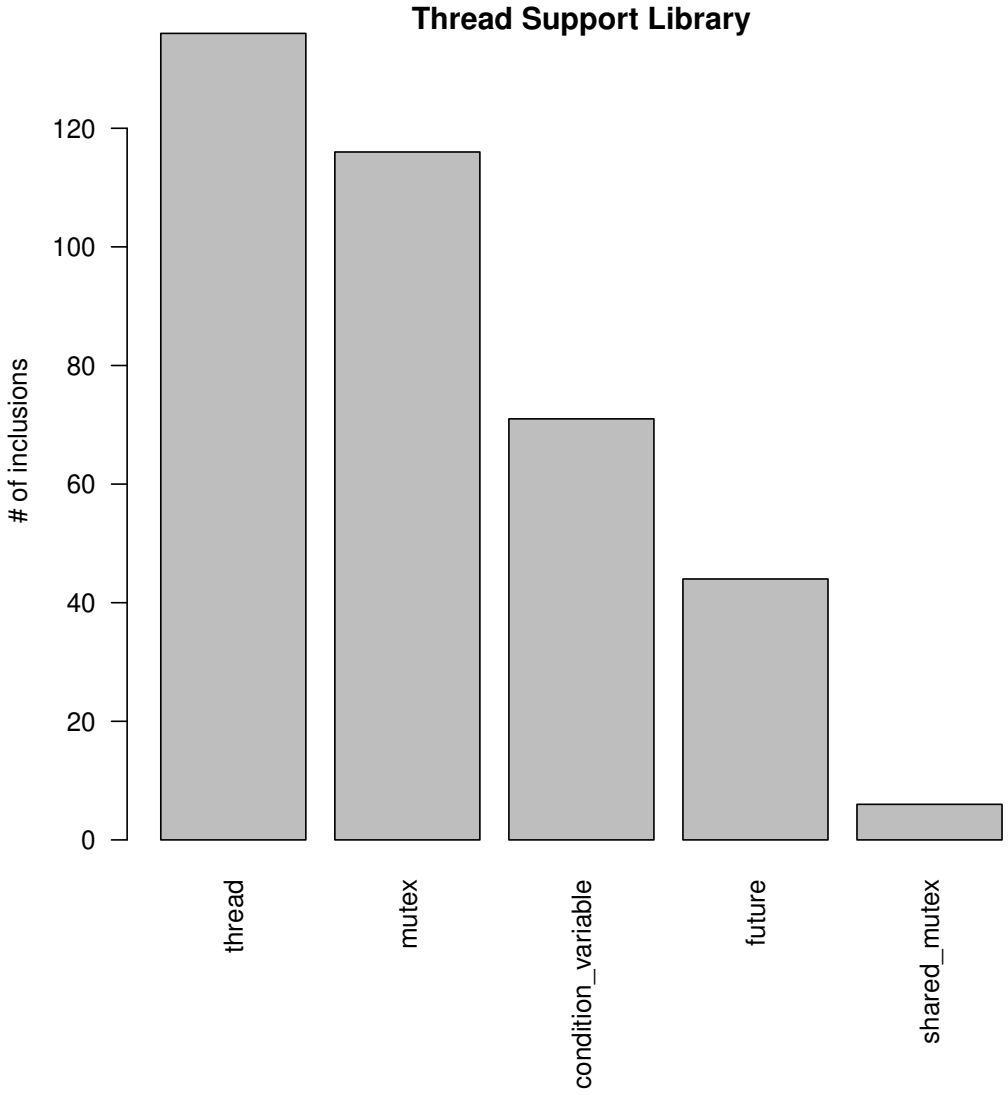

Figure 2.7: Inclusion counts of headers from the thread support module.

Since we are not planning to implement exception handling support in this project, one might think this library module will not be included either. However, while we can turn off exception support through the use of a compiler flag, any project that uses exception related features would fail to compile. This obstacle can be surpassed by redefining exception related keywords such as `try`, `catch` and `throw` by macros.

Even if we redefine these keywords, though, usage of the `throw` keyword that uses an exception object that is either from the standard library or derives one of the standard exception object would still fail to compile. Because of this, it would be beneficial to implement these objects at least in the form of stubs (i. e. implement their interface but have their members do nothing).

2.2.9 Localization Library

The localization module of the standard library provides programs written in C++ with ways to overcome cultural differences such as character sets, currencies, date formats and the textual representation of numbers. These features are acces-

sible through the standard headers <locale>, which contains tools to represent most of the culture specific data mentioned previously, and <codecvt>, which contains functions that can be used to convert strings between different character sets. Note that the latter header has been deprecated as of C++ 17, so when we talk about the importance of this module, we refer to the former header alone.

While the ability to represent data in a culturally appropriate way is a common requirement for contemporary software, HelenOS itself does not offer full localization support and such support is not that important for porting programs to it. That being said, the C++ standard requires other modules (such as the I/O module) to be locale aware and describes some of their operations using types and functions from the <locale> header. While it would be possible to write these parts of the library without using this header, future implementation of this module would require extensive refactoring in the library's source code.

Because of this, even if we do not implement this module we should implement a place-keeping version of it that has the required interface, but performs no or limited logic. By doing this, we make sure that when and if the later implementation is done, no significant changes to the code outside this module are required.

2.2.10 Numerics Library

This module includes, besides wrappers for math related C wrappers, the support for numeric algorithms, pseudorandom number generation, complex numbers and numeric arrays (vectors that have element-wise mathematical operations). As we can see in the inclusion graph in Figure 2.8, the latter two headers are seldom included. The former two headers were included in about a third of the projects, but both numeric algorithms (of which there are only five not counting overloads) can fairly easily be implemented when a ported program requires them.

The ease of implementation of <numeric> stems from the algorithms being in most cases simple iterations over collections performing a mathematical operation. In the case of <random>, the defined API is not as simple, but most of its functions and types serve for distribution and randomness purposes and as such can be substituted with the C standard library facilities at the cost of only skewing the results. These characteristics of the module lead to it having a lower priority than the joint inclusion count of its headers would imply.

2.2.11 Remaining Library Modules

The modules of the standard library we have inspected so far consisted of multiple headers. This meant that in each case we had the possibility to change the priorities on a per-header basis. The remaining modules all consist of a single header and as such do not warrant the need for a closer inspection similarly to what we did in the previous sections. These modules are:

- Algorithms library contains functions that allow their users to provide algorithmic operations on containers. Similarly to the <numeric> header, these often consist of an iteration over an iterator range and applying an operation to each element. This means that, in many cases, these functions can be implemented as they are needed for porting on an individual basis

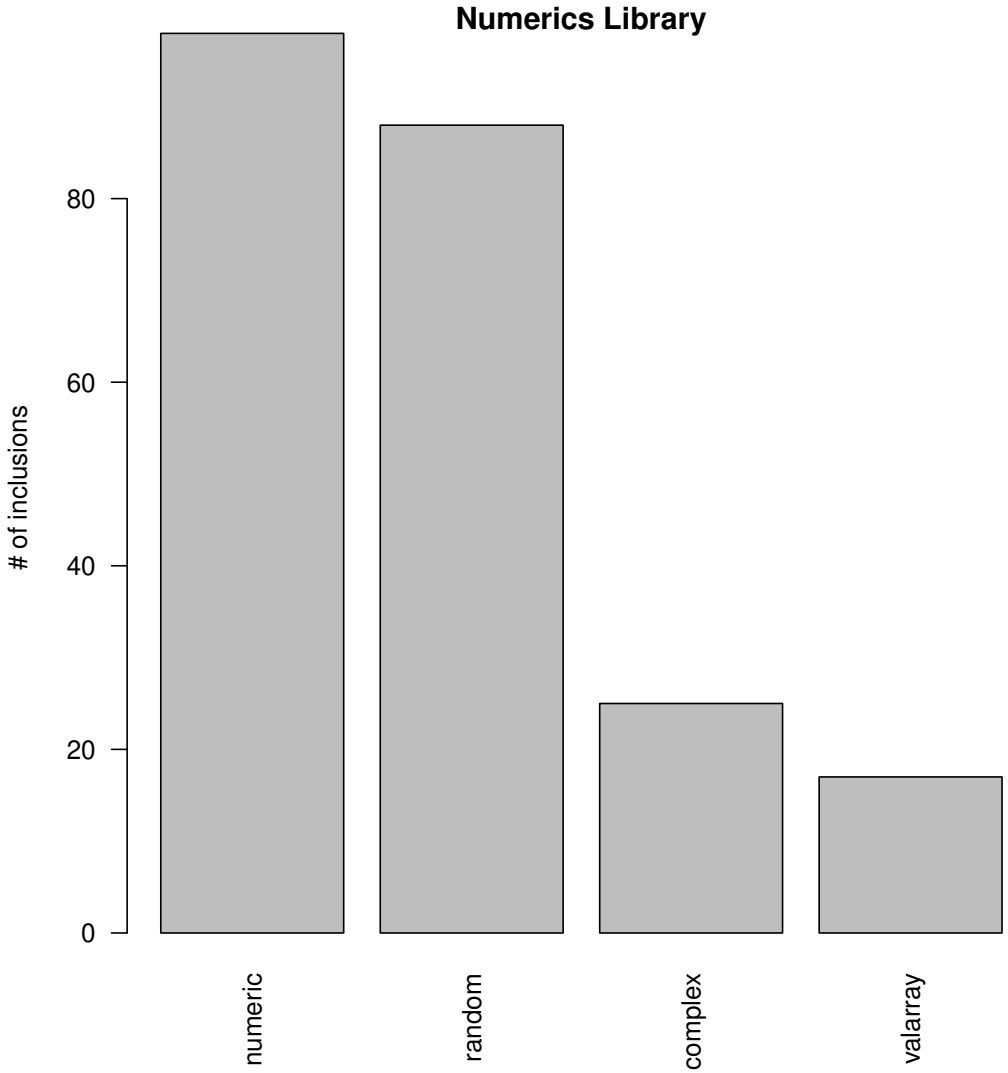

Figure 2.8: Inclusion counts of headers from the numerics module.

in a fairly short amount of time. However, they are often used in different places in the rest of the standard library and as such we decided to apply a lazy approach to the implementation of this module. This means that, while assigning it a low priority, some of the algorithms will be implemented as they are needed elsewhere.

- Atomic operations library provides wrappers around primitive types that allow atomic access and modification to the underlying values. The features of this module are generally implemented through the means of compiler APIs for atomic operations (such as the `atomic_*` functions provided by GCC). Because of this and the relative verbosity of this module, we decided to assign it a lower priority than its inclusion count would indicate.

- Iterators library contains types and functions used to support iteration over containers such as iterator adapters and iterator traits. Many of the containers (and other facilities) in the standard library and as such we think this module (or at least the parts used by the rest of the library) should have a high priority.

- Regular expressions library contains features that allow C++ programs to match and search for regular expressions. This module was the least included of all of the standard library modules we checked. While the reasons for this lowest position are unknown to us, the low inclusion count and the availability of various regular expression libraries (such as Boost.Regex [13]) caused us to keep the low priority of this module.
- Strings library provides the `string` type, which acts as an expandable wrapper around plain C strings (pointer to an array of characters) that manages the lifetime of the underlying array and keeps its length to avoid accidental access to memory behind the end of the underlying array. Not only because of these characteristics has this type or its equivalents from other languages been extensively used. Because of this, we think that it should have a high priority.

2.2.12 Conclusion

Header or module	Count
Containers	1317
I/O	1017
Utilities	804
Threads	329
Strings	229
Iterators	131

Figure 2.9: High priority headers and modules.

Header or module	Count
Diagnostics (stub)	254
Localization (stub)	82
<bitset>	49
<future>	44
<typeindex>	18

Figure 2.10: Medium priority headers and modules.

Header or module	Count
Algorithms (lazy)	231
Numerics	228
Atomics	97
Regular expressions	63
<forward_list>	25
<scoped_allocator>	3

Figure 2.11: Low priority headers and modules.

We have now examined the different modules contained in the C++ standard library in order to determine the priority of each of the modules' implementation for the purposes of porting programs written in C++ to HelenOS. Figures 2.9, 2.10 and 2.11 show the division into three priority groups we have created (along with the summed inclusion counts in the case of modules and inclusion counts in the case of headers).

As one can see, we have modified the priority of five headers that we did not find to share the same level of importance with the other headers in their modules and in three cases we have decided to implement modules either as stubs (i. e. with limited or no functionality just for the sake of compilation) or in a lazy

way (i. e. implementing their functionality as it is needed by other parts of the library).

We have originally set the priorities by the summed inclusion counts of the modules' headers. These counts were over 800 for the high priority group, between 200 and 400 for the medium priority group and under 200 for the low priority group. As one can see, the final priority group memberships violate these ranges quite often. This points out that this initial division was indeed just a heuristic and our closer inspection was justified.

3. Design and Implementation

In this section, we will focus on documenting both the runtime and standard library. However, both the API and the semantics of these libraries are already defined by the standard and the Itanium ABI. Because of this, we will prioritize the documentation of those parts of these two libraries that are not mentioned in either document (and thus might be HelenOS specific). This includes the explanation of some advanced C++ concepts that are needed to understand and possibly maintain the code base should the need arise.

3.1 Template Metaprogramming

One of the concepts that is used heavily throughout the standard library is template metaprogramming, i. e. using C++ templates to generate, modify or remove code at compile time. Its use ranges from simple things such as disabling parts of code unless the template parameters of that code have some specific attribute (e. g. the parameter is integral) to more advanced techniques such as trait types that offer default member types for their template parameters.

However, these advanced topics are commonly left out of the university curriculum when it comes to C++ related subjects, possibly due to their complexity and less than usual usability in the industry. Because of this, the following sections introduce some key template metaprogramming features that are needed in order to efficiently implement the C++ standard library. Note, however, that they are not meant to be a comprehensive introduction to the subject – they serve merely to make any special tricks that may follow more readable. For a more in-depth explanation, see Walter E. Brown's presentation at CppCon 2014 [14] [15], which these following sections are loosely based on.

3.1.1 Metafunctions

One of the reasons that some find template metaprogramming complex might be that by doing it we are essentially abusing a language feature for something it was not intended for. Templates were originally created to serve as means to write generic code and thus avoid code repetition and bad extensibility.

For example, in regular C++ code we would enclose a reusable piece of logic in a function. However, in the world of template metaprogramming (and prior to the addition of `constexpr` functions in the world of compile time programming in general) we do not have the ability to use functions. That is because template metaprogramming is programming with types. Instead of functions, we use so called **metafunctions**. There are two commonly used kinds of metafunctions – those that return types and those that return values.

In Listing 7, we can see both of the aforementioned types of metafunctions. As one might notice, the language does not offer a way for us to return something from a metafunction. Returning here is performed by following conventions, i. e. in the case of a type returning metafunction, we can access the result through the `type` member type alias, and in the case of a value returning metafunction, we can access the result through the `value` static data member. Other than

```
template<class T>
struct add_reference
{
    using type = T&;
};

template<int x>
struct return_double
{
    static constexpr int value = 2 * x;
};
```

Listing 7: Examples of a metafunction that returns a type and a metafunction that returns a value.

adhering to these conventions, nothing distinguishes these metafunctions from ordinary templates.

Through these simple conventions, we can create basic metafunctions. However, in order to write more complex metafunctions, we need the ability to make decisions at compile time. The template language contains three features that help us in this regard – SFINAE, pattern matching and recursion.

3.1.2 Pattern Matching and SFINAE

The ability to use pattern matching in C++ template metaprogramming can be achieved through a rule of the language that causes a more specialized template to be taken into consideration first in the case of partial specializations [2, Section 14.5.5.1]. This rule is used in combination with a characteristic of the language commonly referred to as Substitution Failure Is Not An Error (*SFINAE*), which means that if the compiler tries to substitute a type into a template during resolution and fails, that template (or template specialization) is silently discarded and the compiler tries another one (a less specialized as was previously mentioned). Only when no more templates or template specializations are available to the compiler, the compilation fails.

For example, according to [2, Section 20.7.8.1], the standard `allocator_traits` structure specialized for a given allocator type `A` defines a type alias called `pointer` as follows:

1. `A::pointer` if `A::pointer` is valid and denotes a type
2. `A::value_type*` otherwise

This means that the `pointer` type alias is optional with a default based on the mandatory `value_type` type alias. Achieving this is no simple matter, but we can do so with SFINAE and a concept discovered during the preparation of the C++ 14 standard called `void_t`.

In Listing 8, we can see that `void_t` is a variadic alias template that for any given list of types always acts as an alias to the `void` type. The choice of `void` was

```
// include/__bits/type_traits/type_traits.hpp
template<class...>
using void_t = void;
```

Listing 8: Definition of void_t.

an arbitrary one made by the original author of the type [15]. What it is aliasing does not matter, but it has to be a primitive and predictable type. This template allows us to move some expressions into the process of template resolution by using the type of these expressions (such as e. g. using the `decltype` operator) as its template parameter.

```
template<class A, class = void>
struct get_pointer
{
    using type = typename A::value_type*;
};

template<class A>
struct get_pointer<
    A, void_t<typename A::pointer>
>
{
    using type = typename A::pointer;
};

template<class A>
struct allocator_traits
{
    using pointer = typename get_pointer<A>::type;
    /* ... */
};
```

Listing 9: Implementation of the `pointer` default.

In Listing 9, we can see the implementation of the aforementioned default value for the `pointer` in `allocator_traits`. It does so through the auxiliary metafunction `get_pointer` which is defined as follows:

1. Base template – for a given type `A`, return `A::value_type*`.
2. Specialisation for type `A`, for which `A::pointer` is defined and is a type – for the given type `A`, return `A::pointer`.

If `typename A::pointer` fails during substitution, the compiler discards this specialization and moves to the less specialized base template. SFINAE takes effect here because we injected the alias expression into the template resolution process by using it as a template parameter to `void_t`.

3.1.3 Recursion

Template metaprogramming behaves similarly to functional programming. Because of this, we need to use recursion in combination with pattern matching to achieve more complex tasks.

```
// include/__bits/tuple/tuple.hpp
template<size_t I, class T, class... Ts>
struct type_at: type_at<I - 1, Ts...> {};

template<class T, class... Ts>
struct type_at<0, T, Ts...>
{
    using type = T;
};
```

Listing 10: Finding a type at a given index in a list of types

In Listing 10, we can see a metafunction the standard library uses to find out what type is stored at a given index in a tuple. It does so by decrementing a counter that is initialized as the index we are looking for. At any given step of expanding these templates, the compiler will first try to use the specialization where the index is zero and if that is the case, the metafunction returns the current head of the type list passed to it. If the index is not zero, the compiler falls back to the base template and the metafunction inherits from itself with a decremented index and discard the head of the type list, which represents the recursive call.

3.2 Directory Layout

The runtime is located in the `uspace/lib/cpp`[1] directory. There, the implementation is split into header files (declarations and templates) in the `include` directory and source files (non-templated definitions) in the `src` directory. Since the standard library is heavily templated, most of it is implemented in headers.

The standard requires the standard headers to not have any extension (e. g. <vector> instead of <vector.hpp>). While we do have these headers to conform with the standard, we decided to not put in them their contents and instead use them as middle men. The actual headers lie in the `__bits` directory and diverge from some other implementations (such as the GNU implementation). We did so primarily for two reasons:

- Language support – the lack of extensions would require any developers working on this library to set their editors to interpret files without extensions as C++ source files in order to gain language specific features (such as syntax highlighting).

[1] Note that, unless starting with `uspace/`, any path we mention in this section is relative to that directory for the sake of brevity.

```
// include/functional
#include <__bits/functional/arithmetic_operations.hpp>
#include <__bits/functional/bind.hpp>
#include <__bits/functional/function.hpp>
#include <__bits/functional/functional.hpp>
#include <__bits/functional/invoke.hpp>
#include <__bits/functional/hash.hpp>
#include <__bits/functional/reference_wrapper.hpp>
```

Listing 11: Example of a standard header file in our library.

- Compilation complexity – the standard groups types and functions in headers by their common purposes and areas of use. While this may be needed to ease header inclusion selection for the users of the library, it prolongs the compilation of the library itself as additional types and functions may be included by a source file when in reality only one is needed. An example header file (<`functional`>) can be seen in Figure 11. Here, we have split the header into seven subheaders, each implementing a single functionality or a group of closely tied functionalities. Because of this, when e. g. the `bind` type is needed by another header in the standard library, we do not have to also include the `function` type.

Unlike the header files, the source files are not visible from outside the library and as such their names, locations and contents are not mandated by the standard. We decided to mimic the layout of the `include` directory in the `src` directory with the exception of the top level files already containing the implementation and the `__bits` subdirectory is used for source files that are not directly related to the standard library (e. g. the runtime library or tests).

3.3 Build System

In order to support compiling C++ programs, we had to modify the build system of HelenOS and make it aware of C++ source files to prevent them being compiled as C. Since C++ programs are supported only in the user space as the kernel is entirely written in C and assembly, we had to modify only the user space makefile located at `uspace/Makefile.common`.

The modification itself consisted of two parts: using the correct compiler and the correct compilation flags. Fortunately for us, the HelenOS build system uses GNU Make, which allows the use of pattern rules [16]. These generic rules allow us to use pattern matching (such as by the extension of the file) to apply different rules to different kinds of files.

Listing 12 shows two examples of these pattern rules, one for a C source file and one for a C++ source file. In it, Make applies the indented rules depending on whether the source file being compiled has the `.c` or the `.cpp` extension. This then allows us to use both a different compiler and a different set of compilation flags.

```
# $< = file being compiled
# $@ = output file (the object file)
%.o: %.c
    $(C_COMPILER) $(C_FLAGS) $< -o $@
%.o: %.cpp
    $(CXX_COMPILER) $(CXX_FLAGS) $< -o $@
```

Listing 12: Examples of pattern rules matching C and C++ source files.

Depending on file extensions means that we are limited only to a set of extensions. At the time of writing, these are: `.cpp`, `.cc` and `cxx`. Should the need to add more extensions arise, one can achieve so by simply appending additional rules at the end of the `uspace/Makefile.common` file.

Note that since we started working on this project, the upstream HelenOS repository switched to the Meson build system [17], which eliminated any need for the workarounds required by GNU Make. Currently, a C++ program being compiled on an up-to-date version of HelenOS (including our runtime) simply needs to set the language variable to 'cpp' in its `meson.build` file.

3.4 Runtime Library

As mentioned in Section 2, most of the runtime library was implemented in conformance with the ABI and its semantics were already presented as part of the analysis. The implementation of these parts of the runtime library is part of the sources of the standard library can can be found in the file `src/__bits/runtime.cpp`.

The initialization of global static variables, however, is not defined by the ABI and will be detailed here. It is also not part of the C++ standard library source code as it needs to be executed before a program's `main` function is called.

```
typedef void (*init_array_entry_t)();
typedef void (*fini_array_entry_t)();
extern init_array_entry_t __init_array_start[] __attribute__((weak));
extern init_array_entry_t __init_array_end[] __attribute__((weak));
extern fini_array_entry_t __fini_array_start[] __attribute__((weak));
extern fini_array_entry_t __fini_array_end[] __attribute__((weak));
```

Listing 13: Declaration of the symbols prepared for us by the compiler that refer to the initializers and finalizers.

Since the symbols for both the `.init_array` and `.fini_array` are defined by the compiler, all we had to do to be able to access them was to declare them, which was done in the file `uspace/lib/c/generic/crt/entry.c`, as can be seen in Listing 13. These symbols refer to arrays of pointers to functions that take no arguments and have no return values.

The functions pointed to by elements of the `__init_array_start` array are generated by the compiler and when executed initialize a group of objects (of a certain type). The functions pointed to by elements of the `__fini_array_start`

array, on the other hand, are not needed for the destructors of C++ objects to be called because those are registered to a special `__cxa_atexit` function by the generated initializers. The purpose of these functions is to support the `destructor` attribute (a GCC extension). We have implemented this functionality because the related `constructor` attribute shares its implementation with C++ constructors and we wanted to avoid bringing only half the functionality to HelenOS (more so when it was symmetric to the initializers).

When implementing `__cxa_atexit`, we could not use the C standard library function `atexit` as we need to associate an object to a registered function (that is to be destroyed). We worked around this by creating a secondary mechanism, called `atexit_destructors`, that worked like `atexit` with arguments and the first call to `__cxa_atexit` registers it the first time the function is invoked.

3.5 Standard Library

The following sections focus on describing and explaining our implementation of the C++ standard library. Given that the public interfaces and semantics of the library are already described elsewhere (e. g. in the standard or on reference websites), we will focus primarily on the inner workings of the different types and functions as well as explaining mechanisms and tricks used to implement them.

Because of this focus, the following library modules will be entirely omitted as their implementation is often directly mandated by the standard and they do not require any special techniques:

- Language support library
- Diagnostics library
- Localization library
- Iterators library
- Algorithms library
- Numerics library

Also fully excluded are the two modules we did not implement – Regular expressions library and Atomic operations library.

Similarly treated are comments present throughout the source code. Instead of writing documentation comments for every member and non-member function or type, which can be found in many other publications, the comments focus on behavior not described by the standards, complex parts of the code and auxiliary types and functions not mentioned by the standard at all.

3.5.1 Exception Handling

In Section 2.1.3 we decided not to include support for throwing and catching exception in our runtime library. However, many existing programs written in C++ and even parts of the standard C++ library make the use of exceptions. If we were to only disable exception handling as part of the command line arguments

to the compiler, such programs would fail to compile and our library would have to be completely void of any exception related functionality.

To avoid such situation we decided to overwrite the exception handling related keywords (`try`, `catch` and `throw`) in a manner that allows the code to be compiled and run successfully unless an exception is actually thrown and abort should a throwing scenario be encountered.

```
try
{
    /* ... */
}
catch(const std::exception& ex)
{
    std::cerr << ex.what() << std::endl;
}
catch(...)
{
    std::cerr << "Unidentified object caught, aborting\n";
    abort();
}
```

Listing 14: An example of a common use of the try/catch language feature.

While redefining the keywords with macros works in most cases, there is one issue that had to be addressed. In Listing 14 we can see a common use case of the `try` and `catch` keywords. First, we execute some code in the `try` block and then catch either an exception or an unspecified thrown object. The issue lies in the two uses of the `catch` keyword here – catching a specific object and catching anything.

Listing 15 contains three ways of redefinition we will use to demonstrate the problem. In the first macro, we simply throw away any reference to the caught object (if there is any) and using `if constexpr` disable the compilation of the `catch` block. However, such a macro would fail if the caught object was referenced anywhere in the `catch` block as was in Listing 14 where we printed the message contained in the exception object to the standard error output.

```
/* (1) */
#define catch(expr) if constexpr (false)

/* (2) */
#define catch(expr) if constexpr (expr = {}; false)

/* (3) */
extern std::exception __exception;
#define catch(expr) if constexpr (false)
```

Listing 15: Three simplified ways to redefine the `catch` keyword.

This specific issue is fixed with the second macro where we use the C++ 17 feature allowing us to declare new objects inside the condition of an `if` statement. Unfortunately, this macro will fail in the generic case where we catch any object because that is denoted by three dots which would made the declaration malformed.

Up until now, we have tried to redefine these keywords in such a way that will prevent us from having a need to change software being ported to HelenOS. This, as we have now seen, is not a feasible option as we cannot cover both cases mentioned here. However, the third macro allows us to redefine the keywords in both cases assuming we rename the caught exceptions to the declared external symbol `__exception`[2]. This works because using `if constexpr` with `false` as its condition we guarantee the catch block is not actually compiled and the compiler only requires that symbols we reference in it are actually declared (but not defined, so the `extern` declaration works).

Because of it being able to handle both cases, this is the approach that we used and that can be found in the `__bits/trycatch.hpp` file that contains all the macros necessary for code that makes use of exceptions to compile with exception support disabled (as it is as of the time of writing).

```
/* Original code */
catch(const std::exception& ex) {
    std::cout << ex.what() << std::endl;
}

/* Modified code */
catch(const std::exception& ex) {
    std::cout << __exception.what() << std::endl; // <<- Renamed.
}

/* Expanded code */
if constexpr (false) {
    std::cout << __exception.what() << std::endl;
}
```

Listing 16: Example of how to use our `catch` redefinition macro.

In Listing 16, we can see an example of how this workaround can be used. In the original code we are porting over to HelenOS, the program catches an exception and prints its message to the standard output. In order to work around the disabled exception handling support in HelenOS, we can simply rename the exception wherever it appears in the catch block to `__exception` and the preprocessor will effectively remove the catch block from the compiled code.

[2]Note that this is just a placeholder symbol provided by our library for convenience and one can simply declare the exception outside the affected `catch` block.

3.5.2 Namespaces

HelenOS is not entirely standard conforming, specifically it contains additional symbols in the C standard library include paths (e. g. a struct called `list`) which could potentially collide with our library and user code if they were included to the global (or another) namespace. To avoid any clashes, non-standard symbols included with the C standard libraries are enclosed in a special `helenos` namespace.

Apart from this, there are two more HelenOS specific namespaces, both nested within the `std` namespace. First of these is `std::test`, which contains all the test related types, templates and functions. The other is `std::aux`, which contains implementation specific symbols. Other implementations, such as libstdc++, choose to use two underscores as a prefix for all implementation specific symbols (as the standard marks any symbol with two leading underscores in the name as reserved). We decided to use a nested namespace instead of a naming convention which would mean implementation specific code more well-marked (e. g. because of another level of indentation). The omission of the two leading underscores in the names of these namespaces has no specific reason other than to make it easier to conform with the 80 characters per line limit of HelenOS C style, but because they are namespaces, the addition of two leading underscores is a trivial matter should the need for them arise.

3.5.3 C Library Wrappers

In Section 3.2, we said that headers in the `include` directory do not contain the actual contents mandated by the standard and instead include headers in the `include/__bits` subdirectory. The exception to that statement are wrappers around headers from the standard C library. This is because the wrappers themselves only include their corresponding C headers and import the symbols from these headers to the `std` namespace. This means that language specific support is not necessary for these files and by doing this we avoid an unnecessary inclusion of what essentially is a group of namespace imports.

3.5.4 General Utilities Library

The General Utilities Library, as it name might imply, includes a set of utility types and functions used throughout the rest of the library and elsewhere that are too small to warrant their own library modules. In this section, we will go over the implementation details of some of the more intricate members of this library.

Tuples

A `tuple` is a data structure allowing its users to store objects of different types. It acts similarly to an unnamed structure declared at the point of definition, but it can also be indexed and thus act as a heterogenous array.

While some programming languages implement tuples dynamically (e. g. interpreted or JIT compiled languages), C++ does not provide the `tuple` type as part of the compiler, but rather as part of the standard library – using variadic

templates. In this section, we will go over the principles of the language that allow for tuples to be implemented.

Compile-time integer sequences As previously mentioned, tuples demonstrate an array-like behavior. Using variadic template parameters, we can create a list of types of arbitrary length, but in order to provide indexed member access, we need to be able to assign indices to the inidividual members of a variadic template parameter pack. In order to do so, we will use the standard template `integer_sequence`, or rather its template alias `index_sequence`.

```
template<class T, T... Is> class integer_sequence { /* ... */ };

template<class T, uintmax_t N>
struct make_integer_sequence
{
    /**
     * Recursive to the bottom case below, appends sizeof...(Is) in
     * every next "call", building the sequence.
     */
    using type = typename make_integer_sequence<T, N - 1>::type::next;
};

template<class T>
struct make_integer_sequence<T, uintmax_t(0)>
{
    using type = integer_sequence<T>;
};

template<uintmax_t N>
using make_index_sequence_t =
    typename make_integer_sequence<size_t, N>::type;
```

Listing 17: Simplified implementation of an index sequence.

In Listing 17, we can see the core principle of the implementation of an `integer_sequence`. It is parametrized by two types: `T` representing the type of the members of the sequence and `Is`, which is a variadic list of objects of type `T` that form the sequence.

The instantiation of an `integer_sequence` is done through a metafunction called `make_integer_sequence`, which recursively appends increasing integers to a base empty sequence.

These templates provide us with a way to bind a sequence of integers together and thus allowing us to pass both a variadic list of types and their corresponding indices, which aids us with the implementation of the `tuple` template.

Implementation In order to be able to put different types next to each other in memory as part of a tuple while being able to specify said types at the point

where the tuple is defined, we used inheritance to implement the data structure. However, using inheritance presents us with two obstacles:

- We cannot inherit from primitive types.
- We cannot inherit from the same type twice.

The first problem, preventing us from using a tuple-like `tuple<int>`, has a fairly trivial workaround of storing the primitive types inside wrappers and inheriting from those. Such a practice is already used in some languages to augment primitive types with additional characteristics (e. g. in Java the `Integer` wrapper type is used to give referential semantics to the `int` primitive type).

The second problem is solved by the fact that a tuple must provide an index based access to its elements. In C++ , two specializations of a template are two distinct type so if we embed the index of an element in the wrapper type as its template argument, we guarantee that two tuple elements of different indices are always of different types.

```
template<class T, size_t I>
struct tuple_element
{
    T value;
};

template<class... Ts, size_t... Is>
class tuple_impl: public tuple_element<Ts, Is>...
{
    // ...
};

template<class... Ts>
class tuple: tuple_impl<Ts..., make_index_sequence_t<sizeof...(Ts)>>
{
    // ...
};
```

Listing 18: Simplified tuple implementation.

In Listing 18 we can see a simplified example of the implementation we described above. Each element wrapper, called `tuple_element`, is a template parametrized by the type of the wrapped object and an integer denoting the elements index within the tuple. The `tuple` type itself is parametrized by a list of types it holds and it inherits from an internal type that is parametrized also by an index sequence (using the method described in the previous section) that then inherits from different specializations of the `tuple_element` template based on the types passed and their indices.

Element access Given that a tuple inherits from wrappers around all of its held objects, retrieval of held objects of a given type or at a given index becomes as simple an operation as finding the proper parent type and retyping our tuple to a reference to it.

```
template<size_t I, class T, class... Ts>
struct type_at
{
    using type = /* Type at index I in type list Ts. */;
}

template<size_t I, class... Ts>
auto get(tuple<Ts...>& tpl)
{
    tuple_element<type_at<I, Ts...>, I>& wrapper = tpl;

    return tpl.value;
}
```

Listing 19: Simplified element retrieval implementation.

In Listing 19 we can see an example of how element retrieval based on index would work (element access based on the type of the held object would be analogous). First, let us assume we have a metafunction `type_at` which, given an index and a list of types, returns the type that lies at that index in the given list. We then create the parent type (a `tuple_element` specialization) that corresponds to the specific index in the tuple and create a reference to that type to which we assign our tuple argument. Once we have a reference to the wrapper, we can access its wrapped object and return it.

Operations While the implementation of element access is simplified by the fact that the tuple inherits from the types it holds (or wrappers around those, to be more specific), other operations over the elements of a tuple need to take a more complicated approach – recursion.

Listing 20 shows our implementation of tuple comparison for equality. It is a recursive function that iterates through the elements of two given tuples and at each recursive step compares their elements at indices equal to the current level of recursion. All the other operations are implemented in a very similar manner.

Memory

The memory submodule of the general utilities library defines several functions and structures that describe properties of pointers and pointer-like behaving types as well as tools for memory management. In this section we will explain the inner workings of these traits and a type integral for the manipulation of shared memory in a safe manner.

```
template<size_t I, size_t N>
struct tuple_ops
{ // I < N, we take the recursive step.
    template<class T, class U>
    static bool eq(const T& lhs, const U& rhs)
    {
        return (get<I>(lhs) == get<I>(rhs))
          && tuple_ops<I + 1, N>::eq(lhs, rhs);
    }
};

template<size_t N>
struct tuple_ops<N, N>
{ // Last step of the recursion, I == N.
    template<class T, class U>
    static bool eq(const T& lhs, const U& rhs)
    {
        return get<N>(lhs) == get<N>(rhs);
    }
};
```

Listing 20: Comparison of tuples for equality through recursion.

Traits In the C++ programming language, a trait is a template with (generally) a single template parameter. It contains functions, data members and functions that represent or modify various characteristics of their template parameter.

Rebinding Alongside the so called raw pointers (inherited from C), the traits defined as part of the standard library are compatible with more complex pointers (such as smart pointers that ensure memory deallocation once the pointers leave scope). They do so through so-called rebinds. If `Ptr<T>` denotes a pointer, raw or complex, to a type `T`, then its rebind to type `U` returned by a trait would be `Ptr<U>`.

Default properties In addition to rebinding, to properly support even user defined pointers and allocators the standard defines default values for some pointer or allocator properties. For example, the template `pointer_traits` contains three aliases: `pointer`, `element_type` and `difference_type`. The first alias refers to the template argument of the traits, the second to the type of the pointed to object and the third to an integral type used to represent difference between two pointer values during pointer arithmetic. However, a pointer type does not need to define these aliases in order to be usable by a trait.

In Listing 21 we can see a `pointer_traits` implementation with just these three member aliases. For a raw pointer, i. e. a pointer in the form of `T*` for some type `T`, we specialize the traits as we know all the aliases that need to be defined. However, a more complex pointer may or may not provide these aliases while our traits type has to provide them regardless of their definition in the pointer type.

```
template<class Ptr>
struct pointer_traits
{
    using pointer = Ptr;
    using element_type = /* Ptr::element_type or a default. */;
    using difference_type = /* Ptr::difference_type or a default. */;
};

template<class T>
struct pointer_traits<T*>
{ // Raw pointer specialization.
    using pointer = T*;
    using element_type = T;
    using difference_type = ptrdiff_t;
};
```

Listing 21: A simplified implementation of `pointer_traits`.

Both rebinding and default aliases are implemented through the use of `void_t` as was described in Section 3.1.2 and in Listing 8.

Function Objects

The <`functional`> header provides a generic wrapper of callable objects called `function`. This wrapper is a template with a single template argument that describes the interface of the function – its arguments and return type. For the purposes of calling invocation, such information is enough. However, in order for the compiler to know how much memory an object takes (and thus how much should be reserved inside the wrapper), more information would be required. Unfortunately, this information is not available to us when the wrapper is instantiated, i. e. when the information is needed.

To combat this setback, we have made wide use of a technique called *type erasure* throughout the header. By the term type erasure, we refer to storing the wrapped object in a dynamically allocated piece of memory and pointing to it with a type oblivious (`void*`) pointer. All interaction with the object, such as allocation, copying, destruction and invocation, are performed through a set of callbacks that 'remember' the actual type of the object.

Aside from `function`, which allows us to simply invoke a stored callable object, the header provides also a function called `bind`. This function creates an instance of a type that, besides simple invocation, also allows us to store some arguments in advance that will be passed to the bound callable object upon invocation.

The type returned by the `bind` function, in our library called `bind_t` in the `aux` namespace, stores a tuple with the bound arguments in addition to the callable object which are then passed upon invocation in their respective positions.

Ratio and Time Utilities

The <ratio> header provides compile time representation of rational numbers. It does so by defining a template, `ratio`, the template arguments of which are the numerator and the denominator of the represented rational number. Additionally, the header contains several template aliases that represent mathematical operations over these rational numbers. These are implemented through arithmetic operations over the integral template arguments of the supplied `ratio`s.

The primary use of the facilities provided by <ratio> within the standard library is in the <chrono> header, which provides time related utilities. It does so through two primary types:

- `duration` – represented by:
  - an underlying type
  - conversion rate to a second using the `ratio` template
- `time_point` – represented by:
  - a clock type
  - duration since the clock's epoch

In C and similar languages, functions that take a time related value as their argument generally interpret that value in a predetermined way, such as the amount of seconds for a duration or the amount of seconds since the system epoch for time points. This means that the users of these functions need to know the different formulae used to convert between these units whenever their value is not represented in those exact units. However, since <chrono> provides types representing the different time units, all its users need to do in order to achiev proper conversion is to use type casts. A type cast between two duration types works as follows:

1. A common underlying type for both durations (the one being converted and the one it is being converted to) is determined.
2. A ratio conversion to the requested denominator is performed using the common underlying type.
3. A conversion to the requested underlying type is performed.

Given that a time point conversion is only allowed for time points that share the same clock type, a conversion between two time points is performed by converting the durations from their common epoch.

3.5.5 String Library

The `basic_string` template is implemented as a simple expandable container of a character type specified as a template argument (the `string` and `wstring` symbols in the standard library are just typedefs for `char` and `wchar_t`, respectively).

The implementation is fairly straightforward, and we will not examine it in more depth, but there is one important difference the `basic_string` template has

from e. g. the `vector` template. In order to properly support different character types, the `char_traits` template is used. This template, defined in the same header as the `basic_string` template, has (among others) static member functions for assignment, comparison and conversions of the specific character type and length measurements, comparison, copying and movement of sequences of the specific character type.

```
basic_string& append(const value_type* str, size_type n)
{
    ensure_free_space_(n);
    traits_type::copy(data_ + size(), str, n);
    size_ += n;
    ensure_null_terminator_();

    return *this;
}
```

Listing 22: Example of `char_traits` usage.

In Listing 22, we can see an example member function of the `basic_string` template which appends a C-style string (an array of characters) to the underlying character array. Were we aiming to support only the basic `char` type, we would be able to use a function-like `memcpy` from the C standard library. However, since the argument `n` of the append function refers to the number of characters and `memcpy` requires the number of bytes as its argument, we use `traits_type` (another template argument of `basic_string`, by default this is `char_traits<value_type>`) to provide character type agnostic operations (implemented via template specializations).

3.5.6 Containers Library

The containers library, implemented primarily inside the `include/__bits/adt` directory, provides means of storing, manipulating and operating over collections of data. The different container types implemented within this library module can be classified into three different categories – sequence containers, container adaptors and associative containers. In this section, we will delve further into the implementation details of associative containers, but we will only go over the basic descriptions of the other two categories for the sake of brevity because their designs are rather simplistic due to their sequential nature.

Iterators

An iterator is an object bound to a container that allows its user to iterate over elements of the bound container in a specific manner – most often in the forward and backward directions as well as in a random access way (resembling pointer arithmetic). It can be a simple pointer for array-like containers (e. g. `vector`) or a more complex type that overloads operators commonly used in pointer arithmetic.

The standard (as of C++ 14) defines five types of iterators (enumerated here with a list of basic operators they need to implement):

- `InputIterator` – allows dereference (`operator*`, `operator->`, incrementing (`operator++`, both pre-increment and post-increment, only one pass necessary) and comparison for inequality (`operator!=`).
- `ForwardIterator` – is the same as `InputIterator`, but its increment operators need to allow for multiple passes.
- `BidirectionalIterator` – is the same as `ForwardIterator`, but allows for decrementing (`operator--`, both pre-decrement and post-decrement).
- `RandomAccessIterator` – is the same as `BidirectionalIterator`, but allows for pointer-like arithmetic by overloading the addition (`opetator+`, `operator+=`), subtraction (`operator-`, `operator-=`), comparison (`operator<`, `operator<=` and the `>` alternatives) and indexed access (`operator[]`).
- `ContiguousIterator` – is the same as `RandomAccessIterator`, but requires the underlying data to be stored sequentially in memory.

All of the containers implemented in our library have their iterators also implemented (if needed and their iterators are not just primitive types like a pointer). Their associated iterator types are always denoted by the member alias template `iterator`.

Sequence Containers

Sequence containers store objects in a linear manner, often in the order of their insertion. All sequence containers provide their users with the ability to insert or remove object at or from a specific place in the container, specified either by an index or an iterator. Additionally, most sequence containers also allow for front or back insertion or removal. The basic sequence containers provided by our standard library are:

- `array` – a strongly types alternative to the array data type in C, which in addition to the traditional index based access C-style arrays also provides API similar to other, more complex, C++ containers such as iterators, `swap` or the `get` function (originally designed for the `tuple` template).
- `list` – a bidirectional linked list.
- `vector` – a growing array, allows for index based access as well as access at the back of the sequence.
- `deque` – a bucket based sequence container that allows for constant time insertion both at the front and the back (a double ended queue). It is a bidirectional linked list of arrays (buckets). Each bucket represents a subset of the data stored within the container. Unlike for example `vector`, which reallocates the entire underlying array whenever it needs to grow, `deque` only allocates a new bucket at the beginning or the end of the bucket list. This means that the only time data elements need to be moved is when the user inserts new elements into the middle of the container and insertions at the front and the back can thus operate at constant complexity.

Container Adaptors

Container adaptors are wrapper types that take a container as their template parameter and provide an alternative way of manipulating objects of that container type. The standard library provides three container adaptors:

- queue – a wrapper type compatible with any container that supports insertion at its front, removal from its back and observation at either end (e. g. `list`, `deque`). Provides 'first in, first out' means of storage.
- priority_queue – similar to `queue`, but also requires the wrapped container type to provide an iterator with random access semantics (e. g. `vector`, `deque`). Takes a comparator type as its second template parameter and uses it to provide constant time access to the largest (in the context of the comparator) element stored in it.
- stack – a wrapper around any container that supports insertion, removal and observation of data elements into and from its back (e. g. `vector`, `list`). Provides 'last in, first out' means of storage.

Associative Containers

Unlike sequence containers, which simply store data in themselves generally in the order in which the elements were inserted, associative containers store either keys or key value pairs in an unspecified manner. The C++ containers library defines four different associative containers:

- `map` – stores unique keys and their associated values.
- `set` – stores unique keys.
- `multimap` – stores keys and their associated values, but allows multiple values (different or not) being associated to a single key.
- `multiset` – stores keys and keeps track of the amount of currently stored specific keys by tracking their insertion and removal.

The above four containers, called *ordered* associative containers, allow for a sorted traversal of their stored elements and are implemented as binary search trees in our library. In addition to these, the library also provides an *unordered* variant (named with a `unordered_` prefix) for each and those are implemented in our library as a hash table.

However, given the similarities between the 4 containers (be it ordered or unordered), our library implements only one type for the ordered containers (`rbtree`) and one for the unordered containers (`hash_table`). These two types are then extended by key extractors and policy classes to provide behavior required by the four different containers.

The following sections will go over the implementation of `hash_table` and its associated auxiliary types (the implementation of `rbtree is analogous`).

Underlying data structure All four of the underlying unordered associative containers are specializations of the `hash_table` template in our `std::aux` namespace. It is a hash table implemented by using separate chaining with linked lists. The template itself is fairly simple – it contains the bucket array, the hashing functor, key comparison functor, key extraction functor (described in the following section) and management data such as the load factor or number of buckets. The data management logic is mostly relayed to a policy class specified as the `hash_table`'s template parameter and only logic that is identical to all four containers is contained directly in the `hash_table` template. In order for this one data structure to support all four of the unordered associative containers provided by the standard library, we made use of two generic concepts – storing data in a generic way and using policy classes – which will be described in the following two sections.

Data storage and key extraction Since our `hash_table` supports both key value pairs and just keys, we store those in an abstract way (specified as a template parameter) and use special functors to manipulate them. These functors return the key part contained in these abstract elements – in the case of key value pairs they return the member of the pair that represents the key and in the case of just keys these extractors act as an identity. Note that only keys are needed in the hash table management logic and as such we can leave values to the four specific containers. This mechanism allows the `hash_table` template to manipulate both the `map` and `set` behaviors in a unified way.

Policy classes The difference between the map and multimap containers (and analogously for the set containers) is in the way keys are manipulated. In the multi variant, multiple instances of the same keys (possibly with different associated values) are permitted. To accommodate this we use a policy class, specified as a template parameter of `hash_table`, that provides the logic behind the element manipulation functions (insertions, erasure, search).

```
using table_type = aux::hash_table<
    // template parameters of the specific container
    value_type, key_type,
    // key extractor
    aux::key_value_key_extractor<key_type, mapped_type>,
    // provided by the specific container,
    // most can be specified as template parameters
    hasher, key_equal, allocator_type, size_type,
    // our internal iterator types, for the table and for the buckets
    iterator, const_iterator, local_iterator, const_local_iterator,
    // policy for handling duplicates of keys
    aux::hash_multi_policy
>;
```

Listing 23: An example of how our hash table can be specialized.

Putting it all together In Listing 23, we can see how this `hash_table` template can be specialized to be used as the basis for the `unordered_map` container. The container itself then delegates most of its functionality to the underlying table instance.

3.5.7 Input/Output Library

The Input/Output module of the C++ standard library is the only part of the library that features a heavy object oriented design. The inheritance hierarchy, shown previously in Figure 2.4, consists of the following groups of types, which we will now describe in further detail:

- `ios_base`
- `basic_ios`
- standard I/O streams
- string streams
- file streams

In additions to these, the library also provides additional types and functions, which serve as a way to buffer, format or manipulate the input and/or output, as well as some already instantiated objects from these classes.

Class `ios_base`

The `ios_base` class is the base class for the I/O inheritance hierarchy. It contains (but does not manage) flags of an I/O object that describe its state (e. g. a flag telling the user whether the last I/O operation finished successfully) and current formatting configuration (e. g. to what numeric precision should floating numbers be printed).

Class `basic_ios`

The `basic_ios` class expands on `ios_base` by adding public member functions used to support localization, flag management and synchronization. It is also the first member of the I/O hierarchy that is templated by a specific character type (as well as character traits type, see Section 3.5.5 for an example of their use) and provides relevant type aliases.

Streams

Reading and writing in the C++ I/O library is done through *streams*. A stream is a type that accepts or provides characters from or to some file or device linked to it. As opposed to the C I/O library, which relies mostly on simple format specifiers and primitive types, streams overload the stream (`>>`, `<<`) operators. Since they are overloads, these operator functions can accept more complex structured data types and can be expanded upon by the users of the library (i. e. for any user

defined type, the user can add a new overload for these operators and the C++ I/O library will gain the ability to work with that new type).

There are three groups of types derived from the standard `basic_stream` in the I/O library – standard I/O streams (e. g. `basic_iostream`), files (e. g. `basic_fstream`) and strings (e. g. `basic_stringstream`).

Stream Buffers

Stream buffers are used by streams and are meant to represent the underlying device (`basic_streambuf`), file (`basic_filebuf`) or object (`basic_stringbuf`). In their most basic form they are a set of three pointers to the beginning and the end of a character sequence as well as the current position in the buffer. In addition to these three pointers, a buffer provides member functions that can be used to insert characters into the buffer, retrieve characters from the buffer, put characters back into the buffer and changing the current position within the character sequence.

Formatters and Manipulators

In addition to the types or functions that are directly or indirectly used to retrieve or print characters, the I/O library contains various convenience types that can be used to change characteristics of the streams. Their purpose is to allow for a simple way of changing of formatting and other options of a stream that can be used alongside the manipulated text.

```
// Prints: 1true1
std::cout << true << std::boolalpha
    << true << std::noboolalpha << true;

bool b1, b2{};
std::istringstream s{"0 false"};
s >> b1 >> std::boolalpha >> b2;
// b1 and b2 both have the boolean value of false
```

Listing 24: An example set of I/O manipulators that change how streams handle boolean values.

An example of such manipulators can be seen in Listing 24. There, we use the `boolalpha` and `noboolalpha` manipulators to change the textual representation of boolean values between numerical (0 and 1) and textual ("false" and "true").

All manipulators work in a similar manner. Whenever they are used in a stream operator, the respective stream operator overload is called and receives both the stream and the manipulator as its arguments. It then applies the manipulator (which can be a function or a function object) to the stream.

`cin`, `cout`, `cerr` and `ios_base::Init`

Alongside the various types we have described so far, the I/O library also provides some prepared objects that are instantiated from these types. Namely, the

standard I/O descriptors for standard input, standard output and standard error can be managed by the provided global objects `cin`, `cout` and `cerr`, respectively. The former two are synchronized, which means that they have to reference each other. However, given their nature as global variables, a special object of type `ios_base::Init` is used that performs their initialization (and destruction) in its constructor (and destructor).

3.5.8 Thread Support Library

The threading support in C++ standard libraries tends to be implemented using an underlying API provided by another library (e. g. libstdc++ uses POSIX threads). In the case of HelenOS, the threads used by user space applications are called *fibrils*, which are cooperatively scheduled user space threads. The kernel of HelenOS also provides its own threading API (upon which fibrils are built) that are scheduled. Given that the asynchronous framework, which handles interprocess communication as a layer above the lower level IPC mechanism provided by the kernel, uses fibrils, we decided to primarily focus on fibrils as then the C++ threads would be directly compatible with the asynchronous framework because they expose handles of the underlying threading mechanism through the `native_handle` member function of the `thread` class.

However, in order to not lock us to a single threading API (in case e. g. a different threading API is introduced at a later point in time or we would need to use preemptive threads), the threading library uses an abstraction that acts as a facade to the underlying threading API. Using this approach allows us to change the use of fibrils to the use of a different (but semantically similar) API.

Threading Middle Layer

The threading abstraction layer is created as a template that contains structure definitions for each of the threading primitives (e. g. threads or condition variables) used through it. These structures then contain static member functions for each operation performed on the associated primitives (e. g. initialization or waiting).

In Listing 25, we can see a simplified example of the threading middle layer for the fibril API containing an abstraction for the mutex type and its locking functionality. In order to create a specialization for any given API, we need to create the type that the structure `threading_policy` will be specialized with – in this case called `fibril_tag`, we then specialize the `threading_policy` type and add definitions for the different threading primitives as well as declaring aliases to the types used by the underlying API. Changing which API is used is then done by changing the `threading` type alias to use the appropriate API tag.

Joinable And Callable Wrappers

A typical function that creates a thread in C (such as the `pthread_create` function from the POSIX Thread API or the `fibril_create` function from the fibril API in HelenOS) accepts a callable target of the thread that has the type `int(*)(void*)` – that is, a pointer to a function that accepts one pointer argument that points to an unspecified type – and returns an integral return code signalling either

```
struct fibril_tag;

template<class>
struct threading_policy;

template<>
struct threading_policy<fibril_tag>
{
    using mutex_type = fibril_mutex_t;

    struct mutex
    {
        static void lock(mutex_type& mtx)
        {
            fibril_mutex_lock(&mtx);
        }
        /* ... */
    }
    /* ... */
};

using threading = threading_policy<fibril_tag>;
using mutex_t = typename threading::mutex_type;
```

Listing 25: Simplified version of the threading middle layer.

success or failure. In addition to this function pointer, it then accepts (possibly in addition to other configurational arguments) a pointer that is then passed to the target function.

However, in C++ , a callable object passed to a thread (e. g. a function pointer or a function object) can be of any signature. This means any return type and a variable set of arguments of possibly different types. We have worked around this limitation of the C APIs by creating a C++ specific thread starting function, called `thread_main`, and a wrapper around the callable object passed to `callable_wrapper`. In addition to these two symbols, we had to implement one more called `joinable_wrapper`. This is because fibrils, the threading API we use by default in HelenOS, does not support joining.

Using these three objects, the lifetime of a thread in our C++ standard library looks as follows:

1. An instance of the standard `thread` class is created.
2. Inside its constructor, a callable object is created using a closure created by the C++ lambda functionality.
3. The callable object is wrapped within a `callable_wrapper`, which inherits from `joinable_wrapper`. These two wrappers together manage the execution and joining of the function object passed to the `thread` constructor.
4. An native thread is created using the underlying threading API. It receives `thread_main` as its function to be called argument and the `callable_wrapper` as the single argument to this function. The C++ thread keeps a pointer to the `joinable_wrapper` for the purposes of waiting for its execution to end.
5. Our custom thread starting function is executed in the new thread. It casts its argument to a pointer to `callable_wrapper` (the type of which is the template argument of this thread starting function) and executes it.
6. Once the `callable_wrapper` finishes execution, it signals the fact through a condition variable it shares with its parent (and by that wakes those that used the join operation). If it is in a detached state, it is deleted (this is set through the `joinable_wrapper` pointer kept within the C++ thread object as detaching is a feature closely related to joining).

Header <`future`>

The <`future`> header contains five main types and functions:

- `future`
- `promise`
- `shared_future`
- `async`
- `packaged_task`

All of these use the internal template `shared_state` and its children, defined in `include/__bits/thread/shared_state.hpp`. The basic use of all of these features revolves around storing and retrieving values. However, the standard requires three cases to be taken into account:

- The stored value is of type `R` and is copied or moved in and out of the shared structures.
- The stored value is of type `R&` and what is stored in the shared structures is a reference to the passed object. However, unlike a regular C++ reference, it is stored after the initialization of the shared structure.
- The stored value is of type `void`, which means that nothing is actually stored in the shared structure, but once the value setting functions are called the structure is marked as ready. This makes the `void` specializations suitable for signaling state and synchronization.

Since the `R&` case requires us to allow the user of our types to set the value after initialization but still requires the semantics of a reference, we had to resort to the use of a pointer. However, in order to avoid code duplication, we implemented one version that works for each of the types that are used to retrieve values (e. g. `future`).

```
R get()
{
    /* ... */

    if constexpr (!is_same_v<R, void>)
    { // (1)
        if constexpr (is_reference_v<R>)
        { // (2)
            assert(state_->get());

            // Returns a reference to the pointed to object.
            return *state_->get();
        }
        else // Returns directly the stored value.
            return state_->get();
    }
    // Nothing returned when R == void.
}
```

Listing 26: Retrieval of `R`, `R&` and `void` in one function.

In Listing 26, we can see the implementation of the `get` function of the `future` template. This template uses the `shared_state` structure described earlier, but instead of using `R` as its template argument, it uses the `future_inner_t<R>` meta-function, which works as follows:

- If `R` is a reference type, change it to a pointer.

- If `R` is not a reference type, return `R`.

This makes it possible to store a reference after the shared state was created and to remove that reference from the shared state safely before the shared state itself is destroyed. However, this means that we have to handle values handled by the setter or getter functions in a way that accommodates for this - namely dereference the value (a pointer) stored in the shared state when retrieving a reference and store the address of the value when storing a reference.

We do this by using the `constexpr if` feature of the language, which allows us to perform checks on constant expressions during compilation. In the first `(1)` check, we handle the case of `R` being void and only returning a value from the function if this check fails. In the second `(2)` check, we either return the stored value or the result of dereferencing the stored value depending on whether `R` is a reference type or not.

In order to properly handle these three requirements when it comes to storing values (e. g. in `promise`), the state and exception management logic is stored in a common base template `promise_base` and the three cases are implemented as follows:

- `promise<R>` inherits from `promise_base<R>` and allows for the stored value to be set through the use of a constant lvalue reference or an rvalue reference.
- `promise<R&>` inherits from `promise_base<R*>` and allows for the stored value to only be set through the use of an lvalue reference, storing the address of the referred object.
- `promise<void>` inherits from `promise_base<void>` and does not store anything, only signals that it entered the ready state.

3.5.9 Tests

The library contains a set of 791 tests that are designed to verify that different requirements imposed on it by the standard are fulfilled. These tests use a custom testing framework that provides basic assertion functionality and automatization. While HelenOS already contains a unit testing framework, called PCUT, we needed a more C++ aware tool set including, but not limited to, support for iterator range comparisons and templates.

Including with our library is also a program, called `cpptest`, which when invoked runs all tests and prints a test report to its standard output.

Adding Tests

To add their own tests, one can inherit from the `test_suite` class and implement its two pure virtual member functions:

- `name` returns a string literal containing the name of the test, which is then shown in the output of `cpptest` or any program invoking the test.
- `run` executes the test or tests contained in the class.

A simplified example of such test implementation can be seen in Listng 27. Once created one can add their test class to a test set and execute them (as can be seen in `uspace/app/cpptest/main.cpp`).

```
// include/__bits/test/tests.hpp
class test_example: public test_suite
{
    public:
        bool run(bool report_to_stdout)
        {
            report_ = report_to_stdout;
            start();

            // Our tests:
            test1();

            return end();
        }
    private:
        void test1()
        {
            test_eq(int{}, 0);
        }
};
```

Listing 27: Simplified implementation of a test suite.

Object Lifetime Monitoring

In order to test some features of the standard library, we may need to track the lifetime of an object. An example of such a feature would be smart pointers, where we need to make sure that the owned object gets deallocated when the last owning smart pointer leaves its scope.

To do this, the testing headers include a type called `mock`, which tracks its lifetime (including construction, copying, moving and destruction) through static counter variables. An example of such test can be seen in Listing 28.

```
    mock::clear();
    { // Artificial scope, limits the lifetime of ptr.
        test_eq("constructor cleared", mock::constructor_calls, 0U);
        unique_ptr<mock> ptr = make_unique<mock>();
        test_eq("mock constructor invoked", mock::constructor_calls, 1U);
        test_eq("destructor cleared", mock::destructor_calls, 0U);
    }
    test_eq("mock destructor invoked", mock::destructor_calls, 1U);
```

Listing 28: An example of a test that check an object gets properly destroyed with its owner.

4. Usage

In this section, we will show how to write and build a simple C++ application for HelenOS. In order to do that we need to add a new entry to the `uspace/app` directory. This new directory should contain our source file and a makefile, minimalistic examples of both are shown in Listing 29.

```
// uspace/app/hello/hello.cpp
#include <iostream>

int main()
{
    std::cout << "Hello, HelenOS!\n";

    return 0;
}

# uspace/app/hello/Makefile
USPACE_PREFIX = ../..
BINARY = hello
SOURCES = hello.cpp

include $(USPACE_PREFIX)/Makefile.common
```

Listing 29: A simple C++ Hello World program and makefile.

The source file is the same as it would be for any other system with a C++ runtime, but the makefile needs to be integrated into the HelenOS build system. This is achieved by defining three macros:

- `USPACE_PREFIX`, which in our case is equal to `../..` because our program resides in `uspace/app/hello`
- `BINARY`, which contains the name of the built executable file
- `SOURCES`, which contains a list of source files used to build our executable and in our case this macro contains only the file `hello.cpp` (note that the build system detects C++ projects by one of the `.cpp, .cc and .cxx` extensions and as such one has to use these extensions if they want to write C++ programs for HelenOS[1]).

Additionally, the `uspace/Makefile.common` file needs to be included. For the program to be built with HelenOS, we need to add the path to our new directory to the `DIRS` list in `uspace/Makefile`. In order for the built HelenOS image to contain our program, we need to add the path to the executable we specified in the `BINARY` variable of our makefile to the `RD_APPS_NON_ESSENTIAL` list in `boot/Makefile.common`.

[1] This requirement can be extended by other extensions in `uspace/Makefile.common`

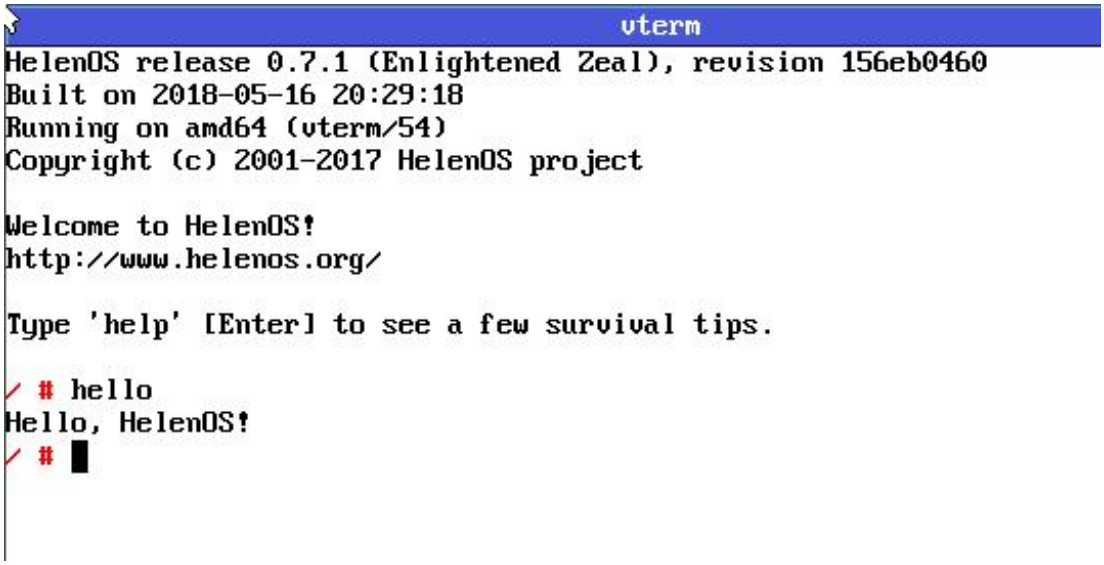

Figure 4.1: Execution of the hello program in HelenOS.

After all of these files are modified or created, we can build HelenOS by executing the `make` command in the root directory and then run HelenOS by executing the `tools/ew.py` utility. Once we boot into HelenOS, we can run our program by typing `hello` into the system's terminal as can be seen in Figure 4.1.

5. Demonstrator

In order to demonstrate the functionality of our C++ runtime, we were searching for a project written in the C++ programming language with the following properties:

- A large project that includes as much of the standard library as possible in order to have proper testing coverage.
- Implemented by a third party to avoid bias when it comes to which features of the standard library are used.
- Accompanied by a set of automated tests which will serve to verify whether our implementation is correct.
- Preferably has little to no dependencies outside the standard library in order to ease the porting process due to the non-standard nature of HelenOS.

In Section 2, we analyzed several open-source projects written in C++ to see which parts of the standard and runtime libraries we would be implementing in order to make our runtime as suitable for porting C++ programs to HelenOS as possible. Our ideal demonstrator project would thus be not one of those we analyzed in order for us to be able to see how the results of our analysis hold against a project not included in it.

5.1 FunctionalPlus

The library we chose, FunctionalPlus [18] by Tobias Hermann, is a C++ library that provides functions and types commonly found in functional languages that allows its users to write C++ program using the functional paradigm. Its single header variant, which we ported to HelenOS, consists of over 16,000 lines of code and includes 43 standard library headers from various areas such as containers, streams, threading and algorithms.

Bundled with the library is also a set of over 1,500 automated tests that will help us verify the correctness of our runtime implementation. These tests also use the only dependency the library has – the doctest C++ testing framework [19]. Fortunately, only a small subset of the macros provided by doctest is used, so this dependency can be avoided by implementing these macros ourselves.

To summarize, this library fulfilled all of our requirements – it is large (around one third of our runtime's size), has only one dependency that can be easily substituted, it is a popular open source library with over 1,500 stars on GitHub and provides nearly twice as many unit tests as we wrote for our runtime and with that brings the total number of tests we can use to verify our runtime's functionality to over 2,200.

5.2 Result

We have successfully ported the demonstrator to HelenOS and made its tests pass with just minor modifications to the FunctionalPlus library itself. The

tests are run by the `cppdemo` application, the sources of which can be found in `uspace/app/cppdemo`.

With regards to headers, out of the 43 required by the library our implementation lacked only the <`forward_list`> and <`atomic`> headers. The former was included only for the purposes of metaprogramming template specializations and the removal of its inclusion had no effect on the library on HelenOS as any user of the library would be unable to use the `forward_list` template. The latter header, <`atomic`>, was never used by the library itself, but rather by one of the tests. This means that even though we were missing two headers the library required, we did not lose any functionality by not implementing them.

In the process of porting FunctionalPlus, we encountered two types of issues:

- Missing features of the runtime that prevented compilation, these could be missing either partially, such as missing template specializations or member functions of existing types, or entirely.
- Bugs hidden in our implementation that caused the tests to fail, including standard non-compliance of our implementation in certain areas.

5.2.1 Missing features

When we started the porting process, missing features were the first kind of issues we have encountered, because they prevent the compilation from finishing successfully. Most of the features that were missing can be split into two groups – mathematics functions and algorithms.

In the first group are three functions that were expected to be in the <`cmath`> header. This header provides functions from the standard C library and as such the absence of these functions was not entirely a fault of our implementation. Rather, the issue was in a design choice made by the HelenOS developers we were not aware of – the functions expected to be in the C <`math.h`> header were declared by the C standard library, but were defined in a different library, called `libmath`. After finding this out and properly linking the additional required library, these issues were resolved.

As for the second group, in Section 2 we have decided to employ a lazy approach when implementing the contents of the <`algorithm`> header. This means that we would implement the different functions when and if they are required by another part of the library. Because of this, our demonstrator program requiring some algorithms our implementation could not provide was to be expected. Following our lazy approach, we ended up implementing 68 different algorithms and once we started porting FunctionalPlus over to HelenOS, we found 12 algorithms to be missing. This means that using our lazy approach, we have implemented 85% of the algorithms required for a full port of our demonstrator.

5.2.2 Bugs

"There are two ways to write error-free programs; only the third one works."

—Alan J. Perlis, Epigrams in Programming

The tests provided by our demonstrator revealed several bugs in our implementation. These ranged from small bugs like off-by-one errors or compilation errors not found sooner due to them happening only in templates with specific template parameters that were never instantiated before to us not understanding the requirements imposed by the standard properly and more significant bugs resulting in crashes.

After fixing all the bugs revealed by the tests to the point when all tests passed, the difference in our code consisted of 1,310 insertions and 232 deletions[1]. With the pull request adding our runtime into upstream HelenOS having slightly over 51,000 lines of code, changes needed by the 1,540 tests provided by FunctionalPlus were contained within 0.45% of the runtime's code.

The changes needed to both fix bugs and implement missing feature in our runtime as well as changes done to the FunctionalPlus library will be provided with this thesis as attachments.

[1]These numbers include the implementations of any missing features.

Conclusion

When we started our work on this thesis, our goal was to implement a C++ runtime that would be capable of porting existing programs written using C++ to HelenOS. In order to do that, we analyzed existing popular open-source projects with the goal to discover a subset of the runtime that could be implemented in the scope of this thesis while allowing us to run ported programs on it.

During our analysis, we have created a priority based system ordering parts of the C++ runtime in order from the most needed to the least needed based on the rate at which they were required by popular projects on GitHub. Then, during the implementation phase of this project, we implemented most of the features deemed to have high or medium priority in the previous phase. Lastly, we successfully ported a reasonably large (in comparison to the runtime itself) software project to HelenOS.

With our demonstrator program ported, the correctness of our runtime was verified with over 2,000 automated tests. Additionally, the porting process has shown only a small amount of missing features required by the ported project, which supported our analysis. Lastly, our implementation has been successfully merged into the upstream HelenOS repository[2] and at the time of merging consisted of over 51,000 lines of code. Because of this, we consider our thesis to fulfill our assignment and goals.

Future Work

As is the case with most software projects, there is still room for improvement and future expansion. As a non-exhaustive list of examples, we present the following project ideas that can expand upon our project:

- Implementing the remaining features of the runtime. These include low level features of the runtime library like exception handling support as well as any missing headers or symbols missing from already implemented headers.
- Modernization of the standard library, bringing it up to the C++ 17 or C++ 20 standards.
- Adding the ability to compile and run C++ program from within HelenOS itself. Two steps are required for that to be possible – our library needs to be added to the HelenOS image and a new enough version of the GNU Compiler Collection that supports C++ 17 (the current version, `4.6.3`, only fully supports C++ 98 with some experimental C++ 11 features) needs to be ported.
- One of the reasons we had for choosing to implement our own runtime from scratch for HelenOS was the desire to implement parts of HelenOS in C++ voiced by some of the developers. As the runtime is now, one would need to deal with C style code and C++ style code in one program whenever HelenOS specific API is used. An alternative to this approach would be to

[2] https://github.com/HelenOS/helenos/pull/41

provide an extension of the standard library (or a whole new library) that contains C++ wrapper types and functions that simplify the use of these HelenOS APIs when utilized within a C++ program. To demonstrate how such a wrapper would work, we will provide a rough idea in the following section.

Example: C++ Approach to Asynchronous Sessions

One of the HelenOS APIs one might want to use from within a C++ program is the Asynchronous Framework, which forms a higher level layer above the lower level inter-process mechanism HelenOS provides. In order to interact with the framework concurrently, the framework provides a concept of sessions [20].

```
async_sess_t *session;
int server_connect()
{
    async_exch_t *exch = async_exchange_begin(NS_SESSION);
    session = async_connect_me_to_iface(exch, SOME_IFACE);
    async_exchange_end(exch);
}

int server_read(int arg, void *buf, size_t bufsize)
{
    async_exch_t *exch = async_exchange_begin(session);

    aid_t req = async_send_1(exch, SOME_SRV_READ, arg);
    async_data_read_start(exch, buf, bufsize);

    async_exchange_end(exch);

    int rc;
    async_wait_for(req, &rc);
}
```

Listing 30: A simplified example of asynchronous session use in C.

In Listing 30, we can see a simplified pseudocode example of how such an asynchronous session might be established and used. First, in the function `server_connect`, we establish an exchange on the naming service session and connect a session to the service. Once the session is established, we can run the second function, `server_read`, concurrently and begin to receive data in an asynchronous fashion by creating a new exchange, sending a request to some server requesting a read operation and then starting the asynchronous read operation itself into the provided buffer. We then close the exchange and can wait for the result to be read (here we wait directly within the `server_read` function for demonstrative purposes, but we could return the request and wait elsewhere).

Once we are done, we need to perform at least two more tasks – handle the received data, including conversions as we receive an array of bytes due to the lack

of generics, and close the session. The former task can lead to excessive amounts of boilerplate code scattered across our program and the latter task may cause us to run to the problem of leaks if we forget to close our sessions.

```
class session
{
    async_sess_t* session_;
    public:
        session(SOME_IFACE): session_{}
        {
            /* Establish the session just like in server_connect()
               in the C example, store it in session_. */
        }

        template<class T> future<T> read()
        {
            return std::async(std::launch::async, ()[]{
                /* Behave just like server_read in the C example,
                   handle errors, then convert and return the result
                   of the exchange once it is available. */
            });
        }

        ~session() { /* End and deallocate session_. */ }
};
```

Listing 31: A simplified example of how an asynchronous session might work in C++ .

In order to demonstrate how the C approach might be simplified by using C++ , Listing 31 presents a simplified pseudocode implementation of a `session` class. This class uses the pair of a constructor and a destructor to manage the lifetime of the session object as well as the establishment and termination of the session itself. With this approach, we are guaranteed that all resources will be safely closed and deallocated when the `session` class instance leaves its scope.

We can simplify the usage of the session itself by utilizing the `future` template, which will allow us to store and manipulate the promised response from the server before it is available and retrieve it once it is. Additionally, since `future` is a template, we can incorporate generic means of conversion to avoid any handling of byte arrays as was necessary in C, because such handling can be hidden under the hood of the `session` class.

Bibliography

[1] HelenOS. http://www.helenos.org. [Online; accessed 2018-05-25].

[2] C++ Standard N4296. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf. [Online; accessed 2018-05-28].

[3] GNU Compiler Collection. https://gcc.gnu.org/. [Online; accessed 2018-05-31].

[4] clang: a C language family frontend for LLVM. https://clang.llvm.org/. [Online; accessed 2018-05-31].

[5] C++ Itanium ABI. https://itanium-cxx-abi.github.io/cxx-abi/. [Online; accessed 2018-05-31].

[6] Meyers, S. *Effective C++*. 3rd Edition. Addison-Wesley, 2005.

[7] Jan Mareš. Port of QEMU to HelenOS. Master's thesis, MFF UK, 2015.

[8] HelenOS Ticket: Add *-helenos-* target to GCC. http://www.helenos.org/ticket/574. [Online; accessed 2021-12-31].

[9] C++ Developer Survey. https://isocpp.org/files/papers/CppDevSurvey-2018-02-summary.pdf. [Online; accessed 2018-11-11].

[10] Bjarne Stroustrup's FAQ. http://www.stroustrup.com/bs_faq.html. [Online; accessed 2018-06-18].

[11] GitHub Blog: Celebrating nine years of GitHub with an anniversary sale. https://blog.github.com/2017-04-10-celebrating-nine-years-of-github-with-an-anniversary-sale/. [Online; accessed 2018-06-22].

[12] Common C++ source file extensions. https://en.wikibooks.org/wiki/C%2B%2B_Programming/Programming_Languages/C%2B%2B/Code/File_Organization#Extensions. [Online; accessed 2018-06-22].

[13] Boost.Regex library. https://theboostcpplibraries.com/boost.regex. [Online; accessed 2018-11-11].

[14] Walter E. Brown: Modern Template Metaprogramming: A Compendium, Part I. https://www.youtube.com/watch?v=Am2is2QCvxY. [Online; accessed 2019-02-08].

[15] Walter E. Brown: Modern Template Metaprogramming: A Compendium, Part II. https://www.youtube.com/watch?v=a0FliKwcwXE. [Online; accessed 2019-02-08].

[16] GNU Makefile Documentation, Pattern Rules. https://www.gnu.org/s/make/manual/html_node/Pattern-Rules.html. [Online; accessed 2019-02-20].

[17] Meson. https://mesonbuild.com/. [Online; accessed 2021-12-31].

[18] FunctionalPlus library. https://github.com/Dobiasd/FunctionalPlus. [Online; accessed 2021-11-12].

[19] Doctest. https://github.com/onqtam/doctest. [Online; accessed 2021-11-12].

[20] HelenOS: Async Sessions. http://www.helenos.org/wiki/AsyncSessions#Entersessions. [Online; accessed 2022-01-01].

Attachments

Attached with the thesis are the following files:

- Files necessary to run our version of HelenOS:
  - `hdisk.img`, which serves as a virtual hard disk for the system.
  - `image.iso`, which serves as the system image that HelenOS is booted from.
  - `run.sh`, which executes Qemu and boots into HelenOS (tested with Qemu version 5.2.0 on openSUSE Tumbleweed).
- Relevant source files:
  - `usr/lib/cpp` contains the sources of both our runtime and standard library.
    - The sources for our tests are located in the `src/__bits/test` subdirectory.
  - `usr/app/cpptest` compililes into the `cpptest` utility, which runs all the automated tests we have written for our runtime.
  - `usr/app/cppdemo` contains the sources of our demonstrator:
    - `fplus.hpp` is a single header version of the FunctionalPlus library.
    - `main.cpp` contains automated tests for the FunctionalPlus library along with auxiliary code to avoid dependencies on the third-party testing library `doctest`.
- List of differences that were needed to be made in order to make our demonstrator work:
  - `diffs/fplus.diff` contains changes done to the upstream header, done mostly due to differences of C++ versions and missing features in HelenOS.
  - `diffs/libcpp.diff` contains all added features and fixed bugs in our library that were needed to make the tests of our demonstrator to pass.
  - `diffs/stats.diff` contains statistics of insertions and deletions from `diffs/libcpp.diff`.
- Scripts and data used in our analysis:
  - `scripts/data` is a directory that contains all header inclusion data used to generate our plots. Specifically, it contains the main `headers.csv` list with inclusion rates for all headers and then files with the prefix `headers_` that accumulate these numbers into groups as discussed in our analysis.
  - `scripts/clone_repos.rb`, `scripts/get_trending_repos.rb` used to get information from GitHub and to clone them.

– `scripts/header_search.rb`, which was used to generate the contents of `scripts/data`.
– `scripts/mkplot.R`, which was used to generate the plots from the contents of `scripts/data`.
– `scripts/repos.txt` lists the GitHub repositories that were used in our analysis and obtained from `scripts/get_trending_repos.rb`.