NTI GYMNASIET
KRONHUS

An internet client library in C++20

Ett internetklient-bibliotek i C++20

Elev: Björn Sundin

Handledare: Victor Frealdsson

Program: Teknik

År: 2020-2021

Preface

The student is currently attending the technology program at NTI-gymnasiet Kronhus. The technology program is university preparatory and includes, among many others, technical subjects such as programming, higher-level mathematics, and physics.

My interest in programming started when I was 13 years old and I first learned the basics of C++ and JavaScript. My first project was programming an RPG game in C++ with my best friend Noah who is an artist, music writer/producer, and animator. In later years, my interest in C++ has grown quite large. I have been digging into all of the details and quirks of the language as well as learned about the features and changes of newer standards.

There are six published C++ ISO standards as of today; C++98, 03, 11, 14, 17, and 20. C++11 was a major update with many new features and improvements that changed the way we wrote code. C++20 is the most recent standard and was published only a few months ago as this is written. This new standard is said to potentially have an even bigger impact on the language.

Learning about all of these new features made me curious and gave me the idea to start this project at the beginning of the school year; a library written from the ground up in C++20, taking advantage of as many new features and techniques as possible. I chose HTTP requests because I had tried a couple of existing libraries for this but was disappointed in their designs and user interfaces (in terms of code; not graphical).

The majority of this work can be read with no knowledge or experience with C++, but some knowledge of programming and general programming terminology is expected. Some parts of section **5.4** are targeted towards readers that have more knowledge of C++ or are willing to do some further reading, as it discusses features of the language in more detail and it is beyond the scope of this text to teach the language. I have tried to refer to pages on CppReference and similar resources where interested readers may read further on the topics.

I would like to thank my best friends and creative souls Noah and Linnéa for being there throughout the years and giving me motivation.

Abstract

This report describes and discusses the development of a piece of software written in the C++20 programming language. The software is a high-level library for sending and receiving data over the internet, primarily over the HTTP protocol. The purpose is firstly to explore how taking advantage of features and changes in newer C++ standards can affect software development in practice. A second purpose is to discuss the usage and effects of different best practices, design principles, and programming paradigms in relation to the project. Another issue that the work explores is unit testing and how good test coverage can be accomplished through design.

The library has specific design goals and restrictions that are presented in the introduction. To give background and context to the project, similar existing works are compared to the idea for this work. The software development process is then described from start to finish and the results presented in terms of the resulting library structure and usage. The most significant results of this work are in the discussion, where topics such as the usage of programming paradigms, modern C++ features, design concerning testability, performance, safety, and error handling are reviewed and examined in relation to the project and with more general conclusions.

Contents

1 Introduction

1.1 Idea

The idea is to write a modern library in C++20 for easily making secured and unsecured HTTP requests.

Some specific aims with the library are:

- A user-friendly, functional design.
- An API that is hard to misuse.
- The code follows the C++ Core Guidelines (Stroustrup and Sutter, 2020).
- Lightweight with minimal dependencies.
- Support for asynchronous requests.
- Support for callbacks for inspecting and/or canceling responses while they are being received.
- Support for Windows, MacOS, and Linux operating systems.
- Free from compiler warnings with all useful C++ warning flags turned on.
- Modern CMake integration.

1.2 Purpose

The purpose of this project was originally to explore how modern language and library features introduced in C++17 and C++20 can affect software and its development in practice. Another purpose is to explore techniques in software development for the abstraction of low-level APIs. Different programming paradigms, design principles, and best practices are discussed in relation to the development of the project.

1.3 Questions to answer

- The library is a set of high-level abstractions on top of low-level APIs. Which APIs are most fitting to build the abstractions from?
- The goal of the project is to design robust interfaces that are hard to misuse. Which language/library features, principles, and paradigms can be used to accomplish this?
- To ensure that the code is correct and that changes do not break the intended behavior of interfaces, the code needs to be testable. What are ways to design the code to accomplish good test coverage?

1.4 Demarcation

Only the HTTP protocol is supported, other protocols like FTP are not implemented. Not all advanced HTTP features are implemented. Only HTTP/1.1 is supported.

2 Background

2.1 Previous works

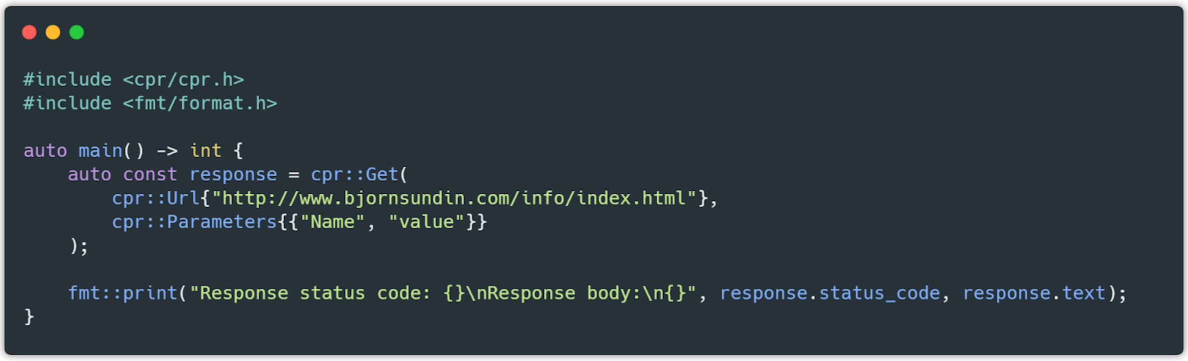
2.1.1 cpr

cpr is an HTTP request library written as an abstraction on top of the popular C library libcurl (Nguyen, 2017). The description on their GitHub page is: "C++ Requests: Curl for People, a spiritual port of Python Requests".

- cpr provides most of the functionality that Cpp20InternetClient aims to provide.
- According to the README.txt file on their GitHub page, cpr requires OpenSSL for making HTTPS requests. OpenSSL is an API mainly for SSL/TLS encryption and decryption (OpenSSL Software Foundation, 2018). A goal of Cpp20InternetClient is to reduce the need for these kinds of dependencies as much as possible, automatically making use of Windows-native APIs on the Windows operating system.
- cpr is built on top of libcurl, a mature C library for making many different types of internet requests (Stenberg, 2020). This has the advantage of making the core functionality stable and trusted by users. However, libcurl is a very big and old library that contains much more than what Cpp20InternetClient aims to support. This project has the aim of being lightweight and having minimal dependencies. Additionally, interacting with less user-friendly, native APIs and implementing more functionality ourselves gives more opportunities to discuss interesting uses of new language features and modern programming practices.
- cpr is written in the C++11 programming language standard and therefore does not make use of newer language features and improvements introduced in C++14, C++17, and C++20.
- Making simple HTTP requests using cpr requires little code and is expressive. The function used to create GET requests is a variadic template that takes any number of parameters for passing things like the URL, request headers, and authentication parameters. These parameters can be passed in any order. See **Figure 2.1** for an example GET request.

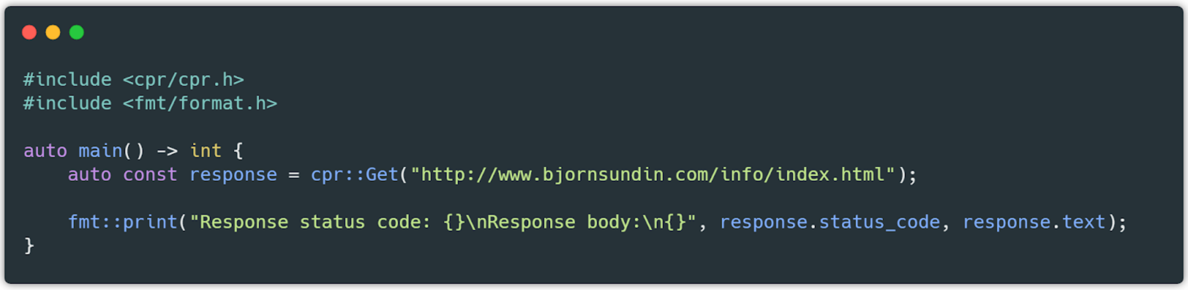
  These variadic template functions are however easy to use incorrectly, as the function declaration says nothing about what types the parameters are required to be. When an object of an unsupported type is passed as one of the parameters to the function, the code still compiles, but `cpr::Get` returns an empty response. An example of code that fails silently in such a way is shown in **Figure 2.2**. It is not apparent from the user's perspective what went wrong. A goal of Cpp20InternetClient is to make interfaces hard to misuse and easy to use correctly, while at the same time hiding low-level details and being expressive. As many errors as possible should be caught at compile-time.
- cpr is not free from many of the useful warning categories provided by compilers like GCC, and can therefore not be built with the -Werror flag (GCC Contributors, n.d.). A goal of Cpp20InternetClient is to compile with -Werror and all useful warning flags turned on.

Figure 2.1. *Example of a GET request with cpr. The fmt library is used for formatted text output.*



```cpp
#include <cpr/cpr.h>
#include <fmt/format.h>

auto main() -> int {
    auto const response = cpr::Get(
        cpr::Url{"http://www.bjornsundin.com/info/index.html"},
        cpr::Parameters{{"Name", "value"}}
    );

    fmt::print("Response status code: {}\nResponse body:\n{}", response.status_code, response.text);
}
```

Figure 2.2. *Example of a GET request with cpr, written incorrectly. This code fails silently and prints an empty response body and a status code of 0.*



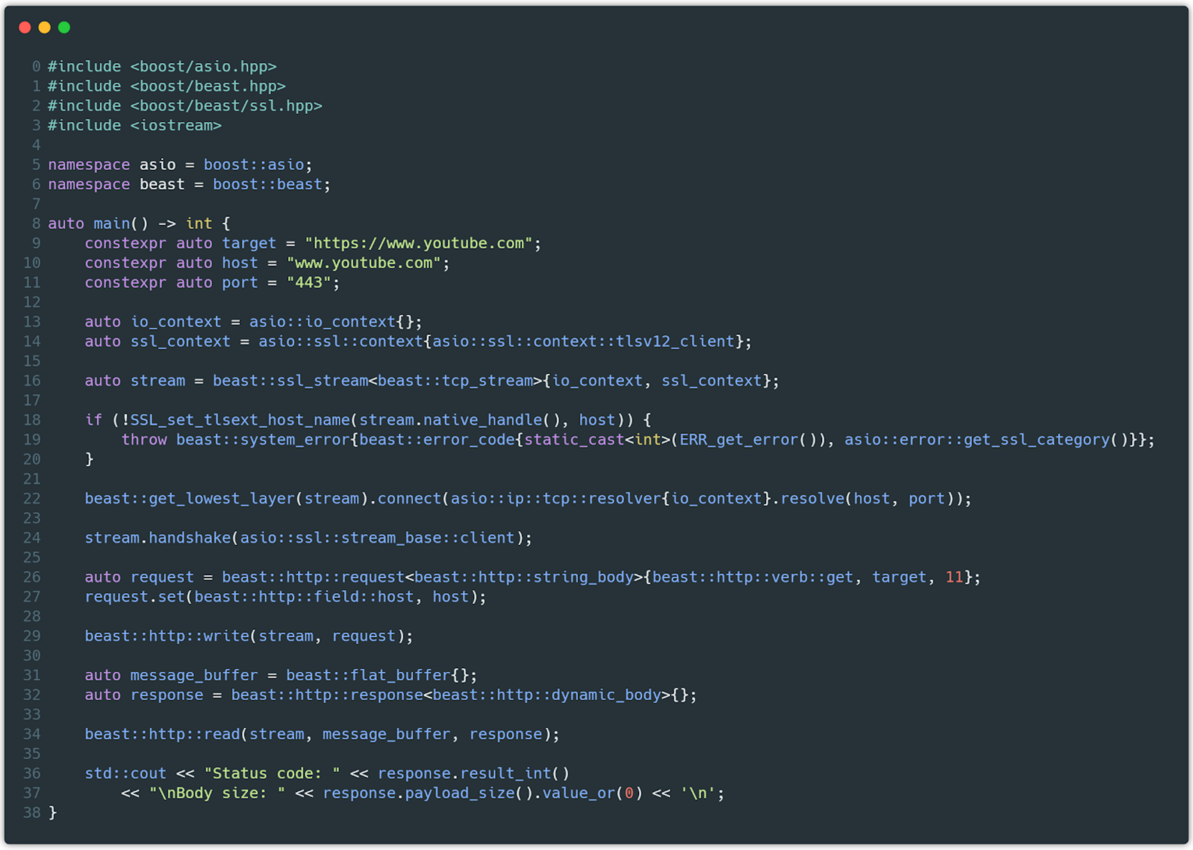
```cpp
#include <cpr/cpr.h>
#include <fmt/format.h>

auto main() -> int {
    auto const response = cpr::Get("http://www.bjornsundin.com/info/index.html");

    fmt::print("Response status code: {}\nResponse body:\n{}", response.status_code, response.text);
}
```

2.1.2 Boost.Beast

Boost is a set of header-only C++ libraries. Boost.Beast is one of these and provides "low-level HTTP/1, WebSocket, and networking protocol vocabulary types and algorithms using the consistent asynchronous model of Boost.Asio" (Falco, 2019). Boost.Asio is a Boost library for low-level network I/O (Kohlhoff, 2020).

- Boost.Beast provides most of the functionality that Cpp20InternetClient aims to provide.
- The library requires OpenSSL for TLS encryption, and like cpr does not take advantage of native Windows APIs.
- Because Boost.Beast is header-only, all of its dependencies are leaked into the global namespace when including the headers, such as the OpenSSL API. Our library will only expose its own API and hide any dependencies in its own translation unit since unwanted names can cause problems.
- The abstraction level of Boost.Beast is relatively low. This means that it is easy to make mistakes in the process of writing code to perform an HTTPS request for instance. See **Figure 2.3** for an example of performing a GET request over HTTPS using Boost.Beast. An advantage with this low level of abstraction however is that it allows for a great level of customization, and abstractions can be built on top of it. In this work, we focus more on ease of use and safety than on being able to have fine-grained control over details, while still being useful for the most common cases.
- Boost.Beast is written in C++11 and does not take advantage of newer library and language features introduced in C++14, C++17, or C++20.
- Boost.Beast is free from warnings when compiled by GCC, which is something this project aims to be as well.

Figure 2.3. *Example of a “GET” HTTPS request using Boost.Beast.*



```cpp
#include <boost/asio.hpp>
#include <boost/beast.hpp>
#include <boost/beast/ssl.hpp>
#include <iostream>

namespace asio = boost::asio;
namespace beast = boost::beast;

auto main() -> int {
    constexpr auto target = "https://www.youtube.com";
    constexpr auto host = "www.youtube.com";
    constexpr auto port = "443";

    auto io_context = asio::io_context{};
    auto ssl_context = asio::ssl::context{asio::ssl::context::tlsv12_client};

    auto stream = beast::ssl_stream<beast::tcp_stream>{io_context, ssl_context};

    if (!SSL_set_tlsext_host_name(stream.native_handle(), host)) {
        throw beast::system_error{beast::error_code{static_cast<int>(ERR_get_error()), asio::error::get_ssl_category()}};
    }

    beast::get_lowest_layer(stream).connect(asio::ip::tcp::resolver{io_context}.resolve(host, port));

    stream.handshake(asio::ssl::stream_base::client);

    auto request = beast::http::request<beast::http::string_body>{beast::http::verb::get, target, 11};
    request.set(beast::http::field::host, host);

    beast::http::write(stream, request);

    auto message_buffer = beast::flat_buffer{};
    auto response = beast::http::response<beast::http::dynamic_body>{};

    beast::http::read(stream, message_buffer, response);

    std::cout << "Status code: " << response.result_int()
        << "\nBody size: " << response.payload_size().value_or(0) << '\n';
}
```

2.2 Concepts and definitions

2.2.1 Cpp20InternetClient

The name that is used to refer to the project in the text. It is also the name of the CMake package namespace.

2.2.2 I/O stream

Something that can both send and receive binary data. This can be a socket, a file, a console user interface, etc.

2.2.3 POSIX

A family of standards that define APIs of UNIX-like operating systems (which include Linux and MacOS) (IEEE, 2018). An example of an API that POSIX defines is the POSIX socket API.

2.2.4 Object files

An object file is what a C++ compiler outputs and what the linker uses as input (Munoz, n.d.). The compiler compiles every translation unit into a binary object file, and the linker links these together, "filling in" any references to compiled code (function addresses) in other object files, and outputting a complete executable or library.

2.2.5 Header files and translation units

A C++ header file usually has the extension `.h` or `.hpp` and contains C++ code that the preprocessor can copy into other C++ files, whether they are other header files or implementation files (Munoz, n.d.). This is done through the preprocessor directive `#include`, which tells the preprocessor to copy the contents of the specified header file into the file that includes it. The preprocessor is a program that is run before any actual compilation of C++ code begins. The output from the preprocessor and input to the compiler is a translation unit.

2.2.6 Implementation files

This is a C++ source file, usually with the extension `.cpp`, that contains code that corresponds to one translation unit and is compiled to an object file. An implementation file can `#include` header files.

2.2.7 SSL/TLS

TLS is a protocol for encryption and decryption of data sent over the internet, providing communication security (IETF, 2018). HTTPS is HTTP encrypted using TLS. SSL is the predecessor of TLS.

2.2.8 TLS handshake

The first step in establishing a secure connection over TLS is performing a handshake (IETF, 2018, Section 4). In this process, different security parameters are negotiated between the client and the server.

2.2.9 Buffer

A buffer is a block of data in memory, consisting of an often large number of bytes (eight bits each). Buffers are used to hold arbitrary data that some part of the program can understand and manipulate for some specific purpose. For example, an image can be stored in a buffer.

2.2.10 HTTP message headers

The headers of an HTTP message comprise a list of (name, value) pairs that provide various types of structured information to the server or client (The Internet Society, 1999, Section 4.2). An example is the `Content-Length` header that indicates the size of the message body in bytes (Section 14.13).

2.2.11 HTTP message body

The body of an HTTP message carries any data that is sent to the server or client (The Internet Society, 1999, Section 4.3). The type of message body data can be specified with the `Content-Type` header (Section 14.17). For example, the contents of an HTML document can be carried in the body of an HTTP response message.

3 Method

3.1 Research methods

Documentation and specifications were referred to when implementing protocols and using APIs. Two primary sources that were used during the development process were Microsoft's API documentation (Microsoft, 2021) and CppReference (CppReference, 2020b). Other sources of information were the HTTP/1.1 specification (The Internet Society, 1999) and the TLS 1.3 specification (IETF, 2018). The book "An Introduction to Modern CMake" (Schreiner, 2021) was used as a resource for learning to write modern CMake. The CMake reference documentation (Kitware, Inc. and Contributors, 2020) was referred to as well. The developer forum StackOverflow (StackOverflow, 2021) was used occasionally for guidance on some issues that arose.

One of the aims of this project was, as mentioned in **1.1**, to follow the C++ Core Guidelines (Stroustrup and Sutter, 2020), which are "[...] a set of guidelines for using C++ well.". These guidelines are an open source, living document created and maintained by Bjarne Stroustrup, who is the creator of C++, and Herb Sutter, a prominent C++ expert. Large portions of this document were read before the development of the library and were kept in mind during development in an attempt to apply the guidelines to the code.

Inspiration was taken from presentations from Cppcon (YouTube, 2021b) as well as from YouTube videos by Jason Turner (YouTube, 2021a). The presentation "Correct by Construction: APIs That Are Easy to Use and Hard to Misuse" by Matt Godbolt (YouTube, 2020) from the "C++ on Sea" conference was also a big inspiration and source of information.

3.2 Work process

The work consisted entirely of concurrently doing research, designing, and programming. The details of this are discussed in the following section.

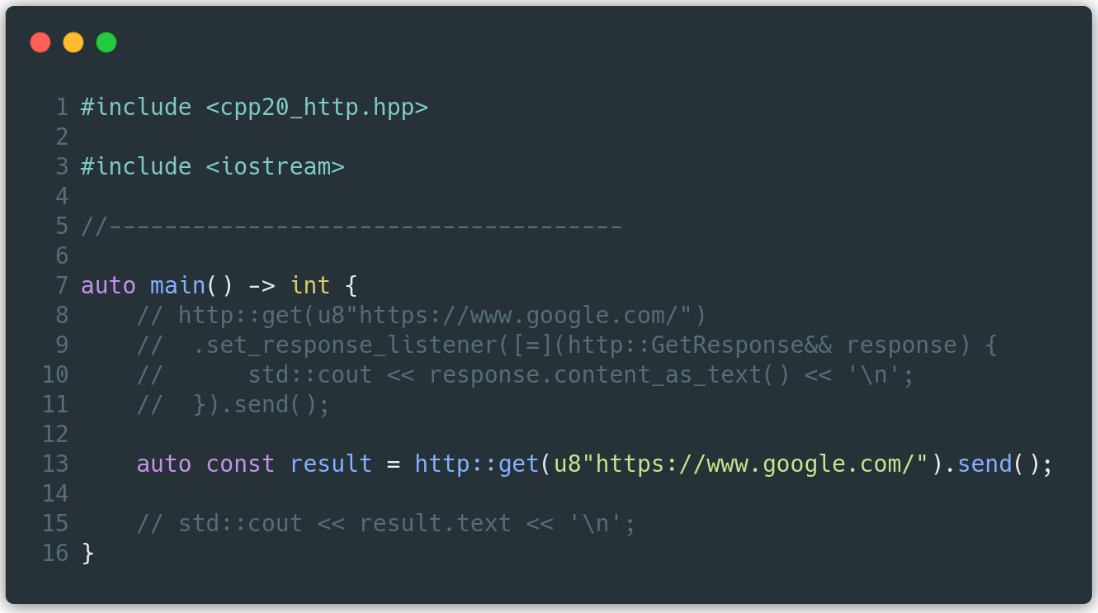
3.3 Software development process

The work was initiated by laying out an initial directory structure, creating a single header file and implementation file for the library code, and adding a small usage concept code snippet as shown in **Figure 3.1**. Starting this way was inspired by the test-driven development technique where test code that checks the correctness of modules is written before the implementation of the functionality (Beck, 2002). The technique can be beneficial because it gives the programmer a clear goal of what the code is required to do and what the interface will look like. As a result, the implementation can be simplified and refactored without worrying about breaking anything from the perspective of the user, since the tests check that the code is correct.

Another example of a (non-strict) application of this technique in the project was the writing of unit tests and usage of a unit testing framework. A unit test is a piece of code that checks the correctness of some functionality. The idea is that the code should be designed such that every component can be tested individually, making bugs quicker to find and improving the stability of whole programs that use these components. The unit testing framework Catch2 was chosen for this project. Catch2 is header-only, relatively small, and written using modern C++ features up until the latest standard (Catchorg, 2021). Catch2 is the second most used C++ unit testing framework, behind Google Test (Jetbrains, 2020). The single-header version of the framework was added to the project and a subdirectory for tests was created.

To build the library, unit tests, and example programs, some kind of build system was needed since running a compiler and linker manually is unsustainable. There are many different build systems, some of which are system-dependent. Two examples of system-dependent build systems with support for C++ are MSVC build tools and XCode. These build systems are targeted towards specific operating systems, compilers, and linkers. To support multiple build systems, a build generator was used. The most used build generator for C++ projects is CMake (Jetbrains, 2020). Using CMake as the build generator was advantageous because it also has a package integration system that allows for simple installation and usage of libraries. Since it is the most widely used build generator, Cpp20InternetClient will be able to integrate easily with a larger quantity of C++ projects. A simple CMake configuration script was written to generate builds of the library, the initial concept program in **Figure 3.1** as well as of the unit testing program.

Figure 3.1. *Initial concept code snippet. This was later developed into a more detailed code example for making a GET request using the library. Note that the name of the library header has changed since then.*



```cpp
#include <cpp20_http.hpp>

#include <iostream>

//------------------------------------

auto main() -> int {
    // http::get(u8"https://www.google.com/")
    //  .set_response_listener([=](http::GetResponse&& response) {
    //      std::cout << response.content_as_text() << '\n';
    //  }).send();

    auto const result = http::get(u8"https://www.google.com/").send();

    // std::cout << result.text << '\n';
}
```

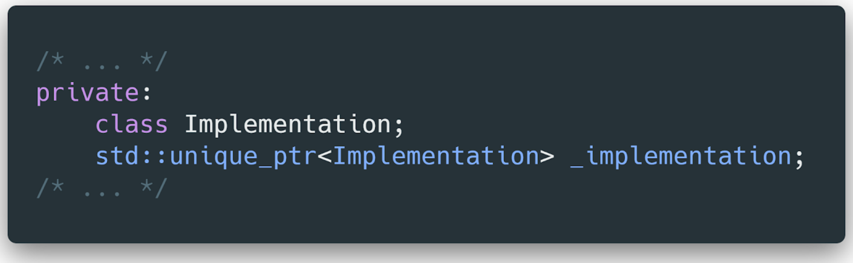
The first functionality that was implemented was the ability to send a GET request over unsecured HTTP, blocking and returning the received response headers and body. The initial development of the library was done on the Windows operating system. Thus, a native Windows API was chosen to provide the most low-level functionality that the library abstracts, assuming there was a corresponding API on Linux and MacOS systems. The first implementation of the mentioned functionality was built on top of an API named

WinINet. WinINet provides a full implementation of FTP, HTTP/1.0 and HTTP/1.1 (Microsoft, 2018a).

However, it turned out that there is no corresponding C or C++ API on Linux and MacOS. After this first working implementation of simple GET requests and responses was finished, the majority of the code was rewritten to make use of a lower-level API that all of the targeted operating systems support equivalent versions of. A class was written that abstracts the only needed platform-dependent functionality; a TCP socket. A socket only provides an I/O interface to a remote computer, in this case, a server. This meant that the HTTP implementation had to be written from scratch.

The PImpl ("Pointer to Implementation") idiom (CppReference, 2020i) was used to separate Windows API and POSIX-based implementations of the `Socket` abstraction. In the PImpl idiom, the interface class (that is declared in the header file and the user interacts with) contains a private forward declaration (CppReference, 2020c) of an implementation class and a `std::unique_ptr` that holds an instance of the implementation. **Figure 3.2** shows what this looks like. This made it possible to write different classes for different implementations of the same functionality, keeping only function declarations in the header file and defining them in the implementation file by forwarding the calls to the inner implementation class instance. Note that this is different from just declaring member functions in a header file and defining them in the implementation file. In the PImpl idiom, the implementation has its own whole class with its own members.

Figure 3.2. *Shows the most important part of the PImpl idiom in the interface class; a forward declaration of an implementation class and a dynamically allocated instance of it.*



```
/* ... */
private:
    class Implementation;
    std::unique_ptr<Implementation> _implementation;
/* ... */
```

Between HTTP and TCP/IP there can also be a layer of encryption, like TLS (Transport Layer Security). HTTP that is encrypted using TLS or its older version SSL is called HTTPS (MDN contributors, 2020). Once the unencrypted socket implementations were finished, an option for TLS encryption had to be added to the `Socket` class for HTTPS support. The Windows Schannel API was used for this on Windows, and OpenSSL was used on Linux and MacOS. Instead of having two different definitions of the implementation class (`Socket::Implementation`), one for Windows sockets and one for POSIX sockets, the functionality was moved into a separate class in the implementation file, `RawSocket`. This was done to better separate the functionality when TLS encryption was added; a `TlsSocket` class was written (also with two implementations for the different platforms) that is built on top of `RawSocket`, only adding the layer of TLS encryption. Then, `Socket::Implementation` only has to hold a `std::variant<RawSocket, TlsSocket>` that either holds a `RawSocket` or `TlsSocket`, forwarding the function calls depending on the type.

HTTP was implemented in the header file to allow for maximum compiler optimization and template parameterization in multiple translation units. The namespace `internet_-client::http` was added to contain the HTTP types and functions provided to the user, and `http::algorithms` to contain types and functions for separating the responsibilities of the implementations of those interfaces (mainly different parsing algorithms). Some examples of these components are `parse_status_line` for parsing HTTP response status lines and `ChunkyBodyParser` for parsing response body data sent with the "chunked" `transfer-encoding` (MDN contributors, 2021). These were not meant for use by other programs, although they could be, and there is no danger in using them. Tests were written for each component of the HTTP implementation.

Support for callbacks and asynchronous requests was implemented and more HTTP request types were added, for example "POST". Multiple example programs were written to perform different types of requests in different ways. Toward the end of the development process, the CMake script was extended to allow for installation and easy integration with other CMake projects.

The library was tested on Windows, MacOS, and Linux. The Windows system was Windows 10 on an internal SSD on a laptop. The Linux system used was Ubuntu version 20.04 and 20.10, on an external SSD booted on the same laptop. The MacOS version was Catalina 10.15.5, on a virtual machine ran on the Ubuntu system using VirtualBox virtualization software.

4 Results

The results presented here are based on version 2.1.0 of the library. The code for this version can be found in the GitHub repository (Sundin, 2021). The library itself consists of 3244 lines of C++ code and the unit tests consist of 577 lines of code. At the time of its release, it could only be compiled by GCC version 10.2.0 or later, not Clang or MSVC. This was because C++20 features were used that the other compilers had yet to implement.

4.1 Library structure

The final directory structure of the project can be seen in **Figure 4.1**. Below is a description of what each directory contains.

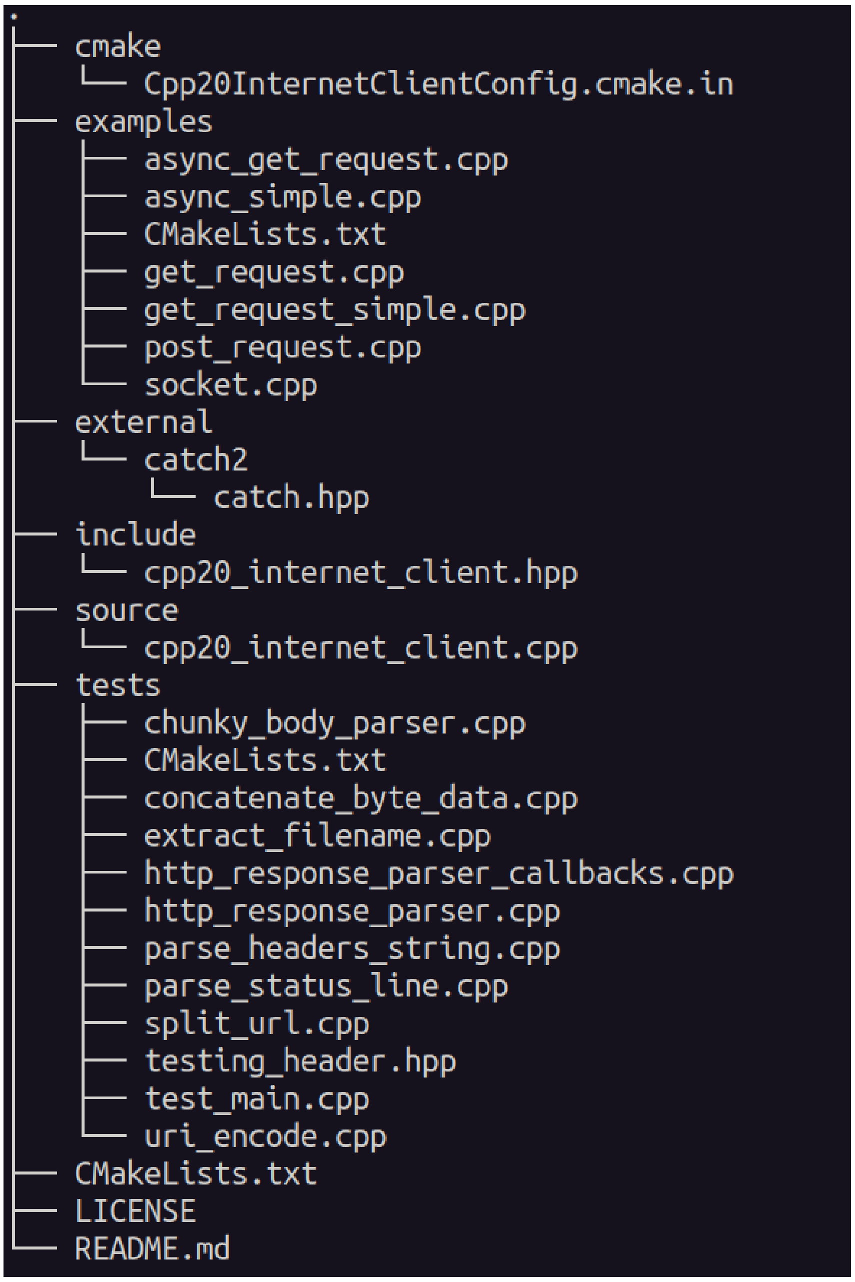
1. **cmake** - This contains a single file, `Cpp20InternetClientConfig.cmake.in`. This is a CMake script that only has the purpose of finding the dependencies of the library during installation.
2. **examples** - These are example programs that use the library to perform different types of HTTP requests using different features of the library. They can be seen as integration tests and as help to other developers wishing to use the library.
3. **external** - This contains smaller dependencies of the library, in this case only the Catch2 unit testing framework header.
4. **include** - Contains the C++ header to be included by programs and by the implementation file.
5. **source** - Contains the C++ implementation file to be built as a static library to be linked to by programs using the library. Only code that depends on external library headers is implemented in this file.
6. **tests** - Contains all of the unit tests for the library. Each file contains one or more unit tests that test a class or function of the library. `test_main.cpp` is a file containing the entry point of the unit testing program.

The library consists of the namespaces outlined below:

```
internet_client {
    errors
    http {
        algorithms
    }
    utils
}
```

`errors` contains types that can be thrown by functions in the library and caught by the user, for example `ConnectionFailed`. `http` contains everything related to the HTTP protocol specifically; for instance, `Socket` falls outside of this namespace. `algorithms` contains all of the lower-level HTTP-related modules that are not expected to be used directly by the user, for example `ChunkyBodyParser`. `utils` contains general utilities that are used by the library. Some of them might be useful for users as well, for example `write_to_file`.

Figure 4.1. *The final directory structure of Cpp20InternetClient.*



```
.
├── cmake
│   └── Cpp20InternetClientConfig.cmake.in
├── examples
│   ├── async_get_request.cpp
│   ├── async_simple.cpp
│   ├── CMakeLists.txt
│   ├── get_request.cpp
│   ├── get_request_simple.cpp
│   ├── post_request.cpp
│   └── socket.cpp
├── external
│   └── catch2
│       └── catch.hpp
├── include
│   └── cpp20_internet_client.hpp
├── source
│   └── cpp20_internet_client.cpp
├── tests
│   ├── chunky_body_parser.cpp
│   ├── CMakeLists.txt
│   ├── concatenate_byte_data.cpp
│   ├── extract_filename.cpp
│   ├── http_response_parser_callbacks.cpp
│   ├── http_response_parser.cpp
│   ├── parse_headers_string.cpp
│   ├── parse_status_line.cpp
│   ├── split_url.cpp
│   ├── testing_header.hpp
│   ├── test_main.cpp
│   └── uri_encode.cpp
├── CMakeLists.txt
├── LICENSE
└── README.md
```

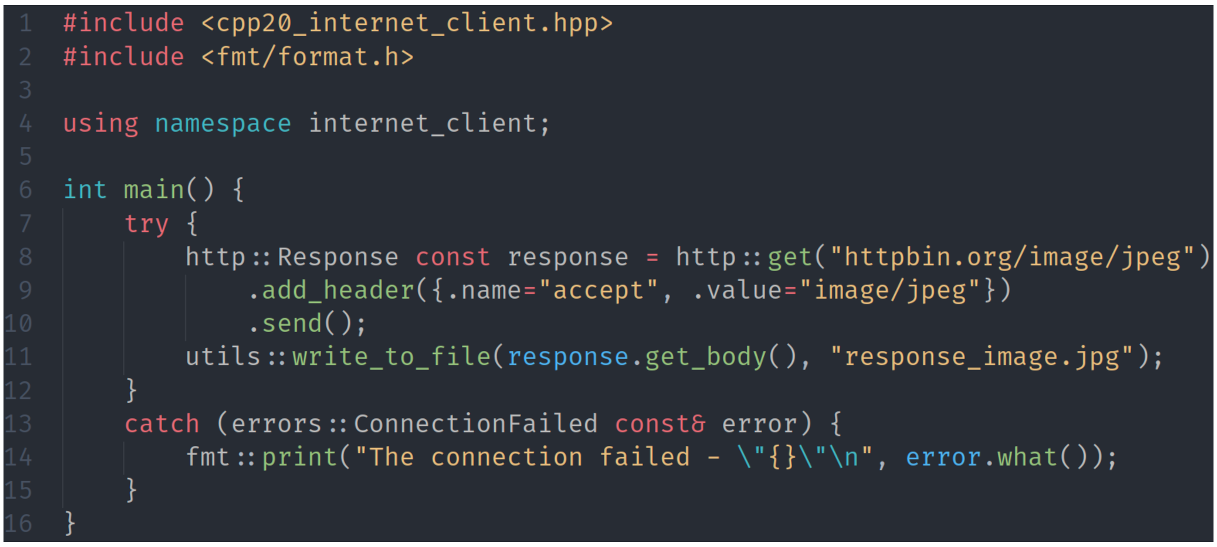
4.2 Library usage

Only two examples of library usage are shown here, and not all provided functionality is demonstrated due to space and time constraints. Refer to the "examples" directory in the GitHub repository (Sundin, 2021) for more usage examples.

Figure 4.2 shows a simple example of a blocking (synchronous) GET request with a custom header field passed as a `http::Header` object and writing the body to a file named "`response_image.jpg`". Any connection errors are handled by printing a description of the error. The output file of the program is shown in **Figure 4.3**. Possible console output from the program in case there is no internet connection is for instance:

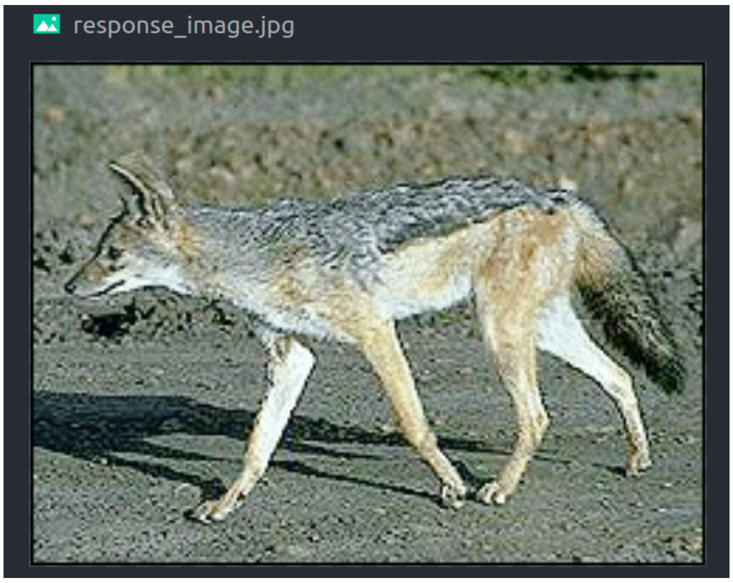
```
The connection failed - "Failed to get address info for socket
creation: Temporary failure in name resolution"
```

Figure 4.2. *Example library usage to perform a simple HTTP "GET" request.*



```cpp
#include <cpp20_internet_client.hpp>
#include <fmt/format.h>

using namespace internet_client;

int main() {
    try {
        http::Response const response = http::get("httpbin.org/image/jpeg")
            .add_header({.name="accept", .value="image/jpeg"})
            .send();
        utils::write_to_file(response.get_body(), "response_image.jpg");
    }
    catch (errors::ConnectionFailed const& error) {
        fmt::print("The connection failed - \"{}\"\n", error.what());
    }
}
```

Figure 4.3. *The output file of the program shown in **Figure 4.2**.*

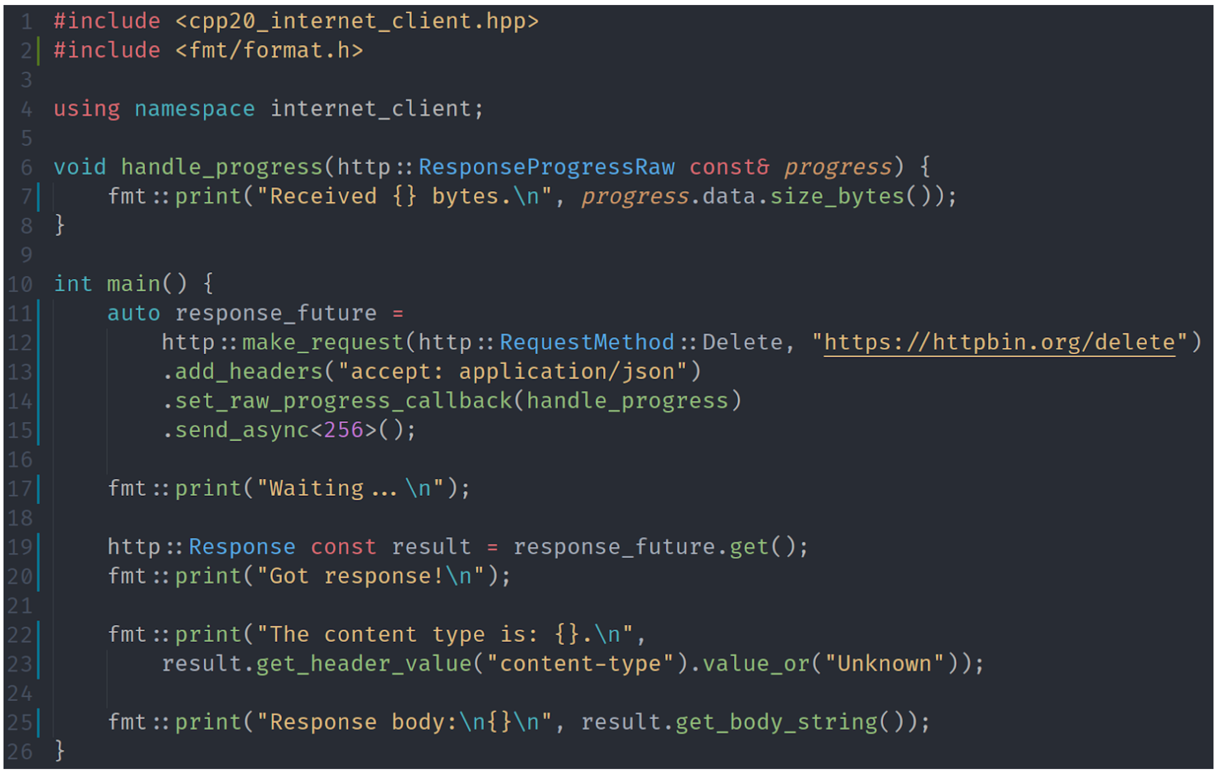


More of the functionality that the library provides is demonstrated in the code snippet in `Figure 4.4`. Here, the function `make_request` is used to create a "DELETE" request over HTTPS. A header is added to specify the type of data to accept in the response from the server, using a string parameter. Four response callback types are supported by the library:

- A raw progress callback can be set with the `Request::set_raw_progress_callback` function and is called whenever one packet of (decrypted) data has been received from the socket. The callback takes a `ResponseProgressRaw` object which contains a reference to the data that has been received so far, an offset to the data that was just received, and a `stop` function to cancel any further receiving and processing of data.
- `Request::set_headers_callback` sets a callback that is invoked when the HTTP headers of the response have been received and parsed. The callback takes a `ResponseProgressHeaders` object which contains an interface for querying header data, a `ResponseProgressRaw` object, and a `stop` function. For example, if the user only wants the headers of the response and does not need to wait for the body to be received and parsed, this callback function can be used.
- `Request::set_body_callback` sets a callback that is invoked when a chunk of the body has been received and processed. The callback takes a `ResponseProgressBody` object which also has an interface to the previously received header data, a reference to the body data that was received so far, an optional total expected body size field that only holds a value if a `content-length` header was included in the response, a `ResponseProgressRaw` object and a `stop` function. An example use case of this callback is to provide user feedback in the form of a progress bar in a graphical user interface or to render what has been received so far of an image that is being retrieved from a website.
- `Request::set_finish_callback` sets a callback that is invoked when the full response has been received and parsed. It takes a `Response` object as the parameter.

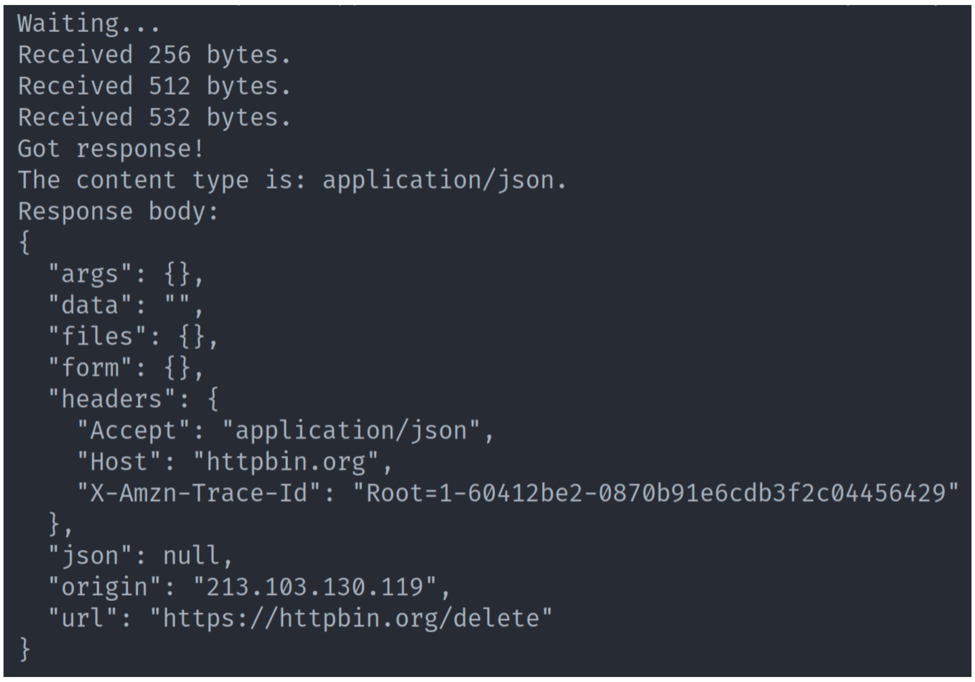
In the example, the usage of a raw progress callback is demonstrated. The size of the data that has been received is written to the console. A lambda functor could have been used instead but a regular function was used in the example for added readability. The request is sent and the response is received asynchronously. A custom receive buffer size is specified as a template parameter to the `send_async` function - if it was left as the default, all of the response data would be fetched at once and the output would not be as illustrative of the callback system. A message is written to the console while the response is received and the resulting response is awaited in the main thread. Lastly, the `content-type` header of the response is printed (or "Unknown" if there was no such header) and the full body data is printed as a string. `Figure 4.5` shows a possible console output of the program.

Figure 4.4. *Example library usage to perform an asynchronous, TLS encrypted "DELETE" request, using callbacks and printing the response body and content-type header value.*



```cpp
#include <cpp20_internet_client.hpp>
#include <fmt/format.h>

using namespace internet_client;

void handle_progress(http::ResponseProgressRaw const& progress) {
    fmt::print("Received {} bytes.\n", progress.data.size_bytes());
}

int main() {
    auto response_future =
        http::make_request(http::RequestMethod::Delete, "https://httpbin.org/delete")
        .add_headers("accept: application/json")
        .set_raw_progress_callback(handle_progress)
        .send_async<256>();

    fmt::print("Waiting...\n");

    http::Response const result = response_future.get();
    fmt::print("Got response!\n");

    fmt::print("The content type is: {}.\n",
        result.get_header_value("content-type").value_or("Unknown"));

    fmt::print("Response body:\n{}\n", result.get_body_string());
}
```

Figure 4.5. *Possible console output of the program shown in Figure 4.4*



```
Waiting...
Received 256 bytes.
Received 512 bytes.
Received 532 bytes.
Got response!
The content type is: application/json.
Response body:
{
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "application/json",
    "Host": "httpbin.org",
    "X-Amzn-Trace-Id": "Root=1-60412be2-0870b91e6cdb3f2c04456429"
  },
  "json": null,
  "origin": "213.103.130.119",
  "url": "https://httpbin.org/delete"
}
```

5 Discussion

5.1 Research methods

The research methods were sufficient because I found the information that I required. I cannot think of any other suitable method of acquiring information for this project. Perhaps taking inspiration from a wider range of sources would lead to a different, potentially more ideal outcome, but this is not reasonably quantifiable.

5.2 Work process

The work process was functional and relatively efficient, although it could have been more structured. The test-driven development technique could have been followed more strictly, where development consists of cycles of first writing unit tests, then implementing the functionality and making sure the tests succeed. This was only done in some cases where it was known ahead of time exactly what the ideal usage of the interfaces would look like. The code was often instead designed with the future writing of unit tests in mind. This did make the code more modular than it would have been without any unit tests at all, and a lot of bugs were quickly eliminated instead of being discovered later in integration tests and through a more painful debugging process. The conclusion is that writing unit tests helped with the work process regardless.

Having a less structured work process meant that code was written in parallel with researching information and designing the architecture of the library. An alternative to this could have been starting with designing the architecture and then working in cycles of research and programming. In this project these three work processes were interdependent, so it is unclear whether following a more structured work process would benefit the result or the efficiency of the process.

5.3 Software development process

5.3.1 Implementing HTTP

Starting the software development on Windows was a disadvantage because of the large number of native networking APIs there were to choose from. It made it possible to use a native API that provided an implementation of HTTP, while there was no corresponding native API on Linux or MacOS. After realizing this, and researching about network sockets, a much better design and abstraction structure were made. Sockets are a basic building block that every operating system provides an API for. More specifically there is a POSIX socket API and a Windows socket API. These are very similar with even some functions having the same names and interfaces.

This meant that I could make a `Socket` class that abstracts the only platform-dependent functionality in the whole library while building all other functionality on top of this, implemented in the header file in a cross-platform manner. This fits well with the way the communication protocols are stacked on top of each other; HTTP only needs an I/O stream to be implemented, and in this case, it is a TCP/IP socket that sends data in packets between a client computer and a server, optionally encrypted with TLS.

Having to write the HTTP implementation from scratch was an interesting challenge and I learned a lot from designing the code for this. The hardest part was implementing the parser for the chunked `Transfer-Encoding` because the data is provided part-by-part in packets of varying sizes. The Single Responsibility Principle was kept in mind while designing the HTTP implementation.

5.3.2 Adding TLS encryption

There were multiple choices for implementing the TLS encryption functionality. It was natural to abstract TLS encryption as part of the `Socket` class, having the option of either an unencrypted socket or an encrypted socket. Implementing TLS from scratch is very complex, as can be seen from the specification (IETF, 2018). It was discovered that there is a native API on Windows for TLS encryption called Schannel, and a corresponding cross-platform API called OpenSSL that is widely used for this purpose (OpenSSL Software Foundation, 2018). Since OpenSSL must be made available on the system separately (although possibly with a C++ library manager like VCPKG), it is only used on Linux and MacOS. On Windows, it is more convenient to use the Schannel API since it is installed together with the Windows SDK. A large difference was noticed in the level of abstraction and ease of use of the two APIs. Implementing TLS encryption using Schannel was by far the hardest part of the whole project.

An example that shows the difficulty in implementing TLS using Schannel was when a fixed size had to be determined for the TLS handshake message receive buffer. This buffer contains a piece of data, a message, received from the server during the handshake. A single function, `InitializeSecurityContext`, is called multiple times to perform a handshake (Microsoft, 2019a). It takes one or more input buffers (most importantly, data received from the server to process) and returns one or more output buffers (most importantly, data to send to the server). The function takes twelve parameters in total, which in itself shows the complexity of interacting with the API.

The output buffer can be allocated by the function, but the input message buffer (data received from the server) needs to be allocated by the caller. Therefore, a maximum message size needs to be determined - unless there is a way to know if the message is incomplete and the buffer needs to be grown. According to the Microsoft documentation, the error code `SEC_E_INCOMPLETE_MESSAGE` is returned if the input message buffer was too small to hold the complete message received from the peer. If this were the case, then the buffer could be grown and more data could be received from the socket, appending it to the buffer and calling the function again.

However, in practice, this was not true. Instead, the error code returned when an incomplete message was passed to the function was `SEC_E_INVALID_TOKEN`. Trying to grow the buffer after getting that error code, calling the function again, and sending the output buffer to the peer causes it to close the connection when trying to read more data afterward. Therefore, a fixed size needed to be determined for the message buffer. It was not clear what this size would be. There is a function `QueryContextAttributes` for querying the TLS message header, message body, and trailer sizes, which would be perfect except for the fact that it only works after the secure connection is established (Microsoft, 2019b).

The only thing the Microsoft documentation for `InitializeSecurityContext` tells us about the size of the message buffer is: "[...] the value of this parameter is a pointer to a buffer allocated with enough memory to hold the token returned by the remote computer." (Microsoft, 2019a). The TLS 1.3 standard specification says: "The record layer fragments information blocks into `TLSPlaintext` records carrying data in chunks of 2ˆ14 bytes or less." (IETF, 2018, p. 78).

Looking at a few implementations of TLS encrypted sockets with Schannel:

1. In an article from Microsoft's documentation (2018b), a sample program in C uses a maximum handshake message size of 12000 bytes.
2. In a version of Chromium by Adobe, the maximum size is $5 + 16 \cdot 1024 + 64 = 16453$ bytes (The Chromium Authors, 2012).
3. In the source of the Curl library, the maximum size is $4096 + 1024 = 5120$ bytes (Stenberg et al., 2021).
4. In the source of the GitHub repository "shells" by Odzhan (2017), the maximum size is 32768 bytes. This is 2^{15}, which is twice as big as the maximum size of a `TLSPlaintext` record chunk.

All of these implementations use different maximum handshake message sizes, which is quite interesting. I decided to follow the TLS specification and use $2^{14} = 16384$ bytes as this is the maximum allowed size of any `TLSPlaintext` record block, which includes handshake messages (IETF, 2018). The buffer is then expanded after the handshake is complete to fit the message header and trailer using the return values of `QueryContextAttributes`.

5.4 Results

5.4.1 Programming paradigms

According to the creator of C++ Bjarne Stroustrup (1995), C++ is not an object-oriented programming language but supports multiple programming paradigms. This was taken advantage of in the project. Cpp20InternetClient uses functional, object-oriented, and generic programming paradigms both together and in different parts of the code to accomplish different goals.

Most of the interfaces to the library are functional. For example, a form of currying is used to build requests. Requests are built in a sequence of function calls instead of with a single constructor taking many parameters. Requests are temporary objects, until for instance `send` has been called and the response object is returned. Furthermore, this response object is immutable and only queries about the response data can be made. Immutable data is also a characteristic of functional programming. Function composition and higher-order functions are also used, especially with the C++20 ranges library discussed in **5.4.2**.

Generic programming means making use of and writing generic functions and classes that are parametrized on types and/or objects at compile-time. In C++, the feature for this is called templates. Templates are a form of static/compile-time polymorphism, as a single interface is provided for multiple types (Bondarenko, 2019). The function call is determined depending on the template parameters at compile-time. Function overloading is also an example of static polymorphism, where the types of parameters determine the function to call. Cpp20InternetClient makes extensive use of templates. For example, the majority of the functions and types in the `utils` namespace are templates, to make them generally applicable to any type that meets the minimum requirements for the type or function. To explicitly constrain these compile-time parameters, C++20 concepts are used. This is discussed further in **5.4.2**.

The project makes use of object-oriented techniques as well. Stroustrup (1995) shows that object-oriented programming is characterized by having abstractions (classes and objects), being able to build upon existing abstractions (inheritance and composition), and making use of dynamic polymorphism (to look up function calls at runtime when interacting with pointers with the type of a base class or interface). To add to this description, it is often said that object-oriented objects contain mutable state, in comparison to objects in functional programming which are more often immutable. While the library makes no use of runtime polymorphism, it does make use of mutable objects and abstract class inheritance.

An example of mutable objects is the `Socket` class, as functions for writing and reading data operate on a single instance. Something interesting about this is that even in a purely functional programming language like Haskell, sockets are object-oriented in a similar fashion (The University of Glasgow, n.d.). I think this has to do with how the underlying socket API in C is designed, or with the concept of a socket in general - it might be hard to translate into a more functional interface.

A better example of an object-oriented design in Cpp20InternetClient might be the parsing algorithms in the HTTP implementation, for example `http::algorithms::ResponseParser`. This class has a single public member function `parse_new_data`, which takes a `std::span` of bytes (a reference to a data block) and returns a `std::optional<ParsedResponse>`, meaning it either returns nothing or a resulting parsed response. The class contains internal parsing state that is modified by the function, and once a full HTTP response has been parsed, the result is returned.

There is one example of inheritance in the library excluding the exception types. The abstract class `http::algorithms::ParsedHeadersInterface` is used to provide an interface to parsed header data for both the response and response progress classes; `Response`, `ResponseProgressBody`, and `ResponseProgressHeaders` (within the `internet_client::http` namespace). The class does not contain any data members, but it contains default implementations of functions as well as a single virtual function for getting the `http::algorithms::ParsedResponse` object that its other functions query.

5.4.2 Usage of C++ language and library features

In this section, the usage of modern C++ language and library features in the project and their advantages are discussed. Only a select set of features are discussed here since very many new features are used.

In C++20, a language feature for modules was added to the standard. Making use of this feature would provide many benefits (ModernesCpp, 2020), but there is not yet full support for this feature in any compiler, nor in CMake. Therefore, the project is planned to make use of modules when the feature is well-supported. Another C++20 feature that the library does not make use of is coroutines (CppReference, 2021b). It may have been possible to make use of them, but there is not yet enough support for them in the standard library and I had not done enough research about the feature to make use of it beyond writing a simple number generator.

C++20 introduces a new, more powerful, and simple way to constrain template parameters, called Concepts (CppReference, 2021a). This is in my experience the C++20 feature that has the largest impact on code. Before concepts, templates could be constrained in a hacky way using something referred to as SFINAE (Substitution Failure Is Not An Error) (CppReference, 2020k). Because of the verboseness of SFINAE techniques to constrain template parameters, they were often left unconstrained. This meant that passing unsupported types as template parameters would result in very obscure compiler errors with references to code inside the template function that failed to compile with the specific type substitution for that template instantiation. With concepts, type requirements are part of the function declaration, which among other things enables more comprehensible and relevant compiler errors. Type constraints are also used by the compiler to select the most appropriate template specializations or function overloads.

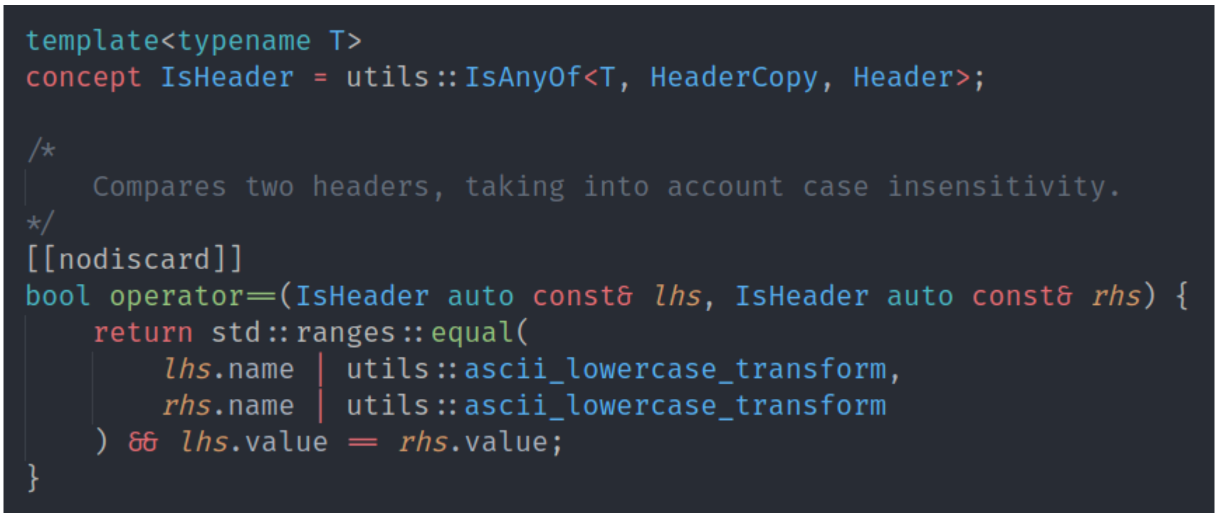
In Cpp20InternetClient, all template parameters are constrained using Concepts, and the library defines its own set of concepts inside the `utils` namespace. One example of an application of Concepts in the library is shown in **Figure 5.1**. Here they are used for an equality comparison operator that works for two different types of header objects; one that holds a copy of the header data (name and value strings) and one that holds string views that refer to data somewhere else. There are many syntax forms for specifying constraints, the form shown in **Figure 5.1** is only one of them. The `IsHeader` concept evaluates to true for types that are either `HeaderCopy` or `Header`. It is defined using the `utils::IsAnyOf` concept shown in **Figure 5.2**, which uses a C++11 parameter pack and C++17 fold expression (CppReference, 2020g) to evaluate whether a type is equal to one of any number of types. Without templates and concepts, the code for the header comparison operator would have to be duplicated four times for all combinations of `HeaderCopy` and `Header` types.

C++20 also comes with a new functional "ranges" library (CppReference, 2021d) as well as versions of the standard library algorithms that are constrained using Concepts, with overloads that take ranges instead of iterator pairs as parameters (CppReference, 2020d). The name field of an HTTP header is case insensitive and only within the ASCII character set (The Internet Society, 1999, Section 4.2). This is taken into account in the comparison operator, using the ranges library to compare the names case-insensitively. A transform is applied to both names that converts them to lowercase, and the constrained algorithm `std::ranges::equal` then compares them. The lowercase transform view is defined in **Figure 5.3**, using the lazily evaluated view and higher-order function `std::views::transform`.

The library contains no usage of regular for-loops; only constrained algorithms, views, and range-based for loops are used. This makes the code more expressive, easier to understand and there is less space for bugs than the equivalent hand-written versions with "raw" for-loops (off-by-one errors, buffer overruns, signed/unsigned mismatches, integer over-/underflows, etc.). The performance cost of using these abstractions is small, as the compiler optimizes them well and all views are lazily evaluated.

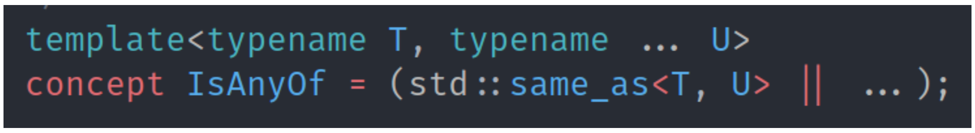
C++17 introduces a new `[[nodiscard]]` attribute for classes and functions, which makes the compiler warn about discarded values (CppReference, 2020a). This attribute is applied to all pure functions for which it is with full certainty a mistake to ignore the return value. An example is any comparison operator, such as the one in **Figure 5.1**. If two objects are compared without using the return value, it is with full certainty a mistake. There are places where the attribute has a bigger impact on making interfaces harder to use incorrectly. An example is `http::Request::send`. Being forced to use the return value makes the attempted usage of the function self-documenting; the result is not queried from the `Request` object, but rather the state is transformed into another interface in a functional fashion.

Figure 5.1. *An example that shows usage of C++20 Concepts, standard ranges library, and the `[[nodiscard]]` attribute in the project.*



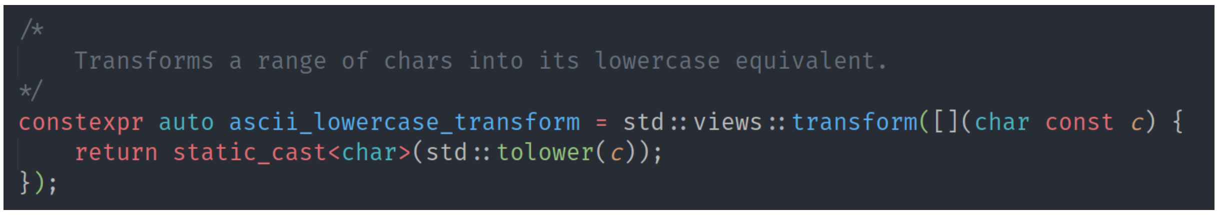
```cpp
template<typename T>
concept IsHeader = utils::IsAnyOf<T, HeaderCopy, Header>;

/*
    Compares two headers, taking into account case insensitivity.
*/
[[nodiscard]]
bool operator==(IsHeader auto const& lhs, IsHeader auto const& rhs) {
    return std::ranges::equal(
        lhs.name | utils::ascii_lowercase_transform,
        rhs.name | utils::ascii_lowercase_transform
    ) && lhs.value == rhs.value;
}
```

Figure 5.2. *The definition of the `utils::IsAnyOf` concept, using C++17 fold expressions and C++20 `std::same_as` concept.*



```cpp
template<typename T, typename ... U>
concept IsAnyOf = (std::same_as<T, U> || ...);
```

Figure 5.3. *The definition of `utils::ascii_lowercase_transform` using the `transform` view from the C++20 ranges library.*



```cpp
/*
    Transforms a range of chars into its lowercase equivalent.
*/
constexpr auto ascii_lowercase_transform = std::views::transform([](char const c) {
    return static_cast<char>(std::tolower(c));
});
```

Another feature that is used as a way to enforce interfaces is reference-qualified member functions. Adding `&&` to the end of a member function declaration specifies that the object that the function is called on must be temporary (CppReference, 2020h). This is used on all functions of the `http::Request` class. The data of a request object is moved once the request is sent, so the object should be discarded at that point. Adding the rvalue qualifier to the member functions enforces this.

In the C language, memory and other resources need to be freed/destroyed/closed manually with function calls, while C++ provides automatic resource management in the form of the perhaps unintuitively named RAII (Resource Acquisition Is Initialization) idiom (CppReference, 2020j). RAII objects hold some resource that is automatically freed when the object goes out of scope. C++11 move semantics play a big part in this, as resources can be moved between objects efficiently without any allocations or deallocations. This allows the lifetime of resources to not be restricted to a single variable scope. All of the APIs that the library abstracts are written in C. To avoid the common bugs associated with manual resource management, RAII should be utilized with these APIs as well. There is a less-known feature in C++20 that is utilized heavily in the project to help with this: lambdas in unevaluated contexts (Dionne, 2016).

Using lambdas in unevaluated contexts, a new lambda type can be specified as a template parameter directly with the `decltype` keyword. This is utilized to easily designate a custom deleter for `std::unique_ptr` types, as seen in **Figure 5.4** where the function `BIO_free` must be used to free BIO pointers in the OpenSSL API. Often in C APIs, a manually managed resource comes in the form of a "handle" that is an integer ID instead of a pointer. For these handles, a `std::unique_ptr` cannot be used. Therefore, a similar type was written for arbitrary handle types and arbitrary "invalid handle" values, named `UniqueHandle`, in the `utils` namespace. **Figure 5.5** shows an example of an instantiation of this class template to define a `SocketHandle` type for POSIX sockets. The first template argument is the underlying handle type, the second is the (required) custom deleter type, and the third is the value used for invalid handles, which in this case is -1. These template parameters are constrained using C++20 Concepts, as shown in **Figure 5.6**. The underlying handle must be trivial and the deleter must be an invocable that takes the handle type as a parameter. The third parameter does not have a type constraint as it is an object and not a type.

Figure 5.4. *Usage of lambdas in unevaluated contexts to directly specify a custom deleter for a* ***`std::unique_ptr`****.*

```cpp
using UniqueBio = std::unique_ptr<BIO, decltype([](BIO* x){ BIO_free(x); })>;
```

Figure 5.5. *Example template instantiation for the* ***`utils::UniqueHandle`*** *class template.*

```cpp
using PosixSocketHandle = int;

using SocketHandle = utils::UniqueHandle<
    PosixSocketHandle,
    decltype([](auto const handle) {
        if (::shutdown(handle, SHUT_RDWR) == -1) {
            utils::throw_connection_error("Failed to shut down socket connection");
        }
        ::close(handle);
    }),
    PosixSocketHandle{-1}
>;
```

The C++ Core Guidelines (Stroustrup and Sutter, 2020) recommend to avoid passing pointers and lengths as separate parameters to functions taking a block of data or an

Figure 5.6. *The type constraints for the* ***utils::UniqueHandle*** *class template.*

```
template<IsTrivial _Type, std::invocable<_Type> _Deleter, _Type invalid_handle = _Type{}>
class UniqueHandle {
public:
```

array, which is very common in older code. Their suggested alternative is to use a span abstraction that holds these as data members. This type can also calculate the length automatically if constructed with an array or any other sized range, eliminating potential bugs from having to manually specify the length. In C++20, this type was added to the standard library. Cpp20InternetClient makes extensive use of `std::span` wherever a view over any contiguous data is needed. The `std::string_view` type introduced in C++17 is similar to `std::span` but is used for immutable references to strings or parts of strings. The library also utilizes this type.

Common bugs in C++ programs stem from forgetting to initialize variables. The values of local objects that are not initialized and that do not have a default constructor that initializes them properly, like trivial types, are indeterminate (CppReference, 2020f). C++11 introduces the `auto` keyword for type deduction (CppReference, 2021c). If the `auto` keyword is used in the declaration of a variable, it must be initialized. Prominent C++ expert Herb Sutter (2013) advocates the use of the Almost Always Auto (AAA) style and presents good reasons to follow it, the strongest being that it is impossible to forget the initialization of variables. Using the AAA style also means that all local variables are declared with the type on the right-hand side of the declaration.

When this style was first adopted, it had some drawbacks that made it undesirable in cases where the type has an expensive move constructor, and impossible if it has no move/copy constructors at all. Since C++17, copy/move construction is required to be elided in certain cases (CppReference, 2020e). This made both initialization styles in **Figure 5.7** equivalent; `Type` is not required to have a move or copy constructor anymore. Therefore, there are no drawbacks to using this style today. The only place where `auto` is not used in local variable declarations in the project is when the intention is to initially leave the variable uninitialized, perhaps because it is immediately initialized by a function from a C API.

Figure 5.7. *Two variable initialization styles that are required to be equivalent since C++17.*

```
// 1.
Type const object{};

// 2.
auto const object = Type{};
```

5.4.3 Error reporting and handling

The C++ Core Guidelines (Stroustrup and Sutter, 2020) recommend using exceptions instead of error codes for reporting errors and provides strong reasons for doing so. Therefore, the library uses exceptions to report errors to the user. The namespace `internet_-client::errors` contains a collection of error types that are thrown by the library. This collection is very small, there are only two error types. These are `ConnectionFailed` and `ResponseParsingFailed`, and they inherit from `std::exception` to conform to the same common interface that all standard library exceptions conform to. `std::exception` only has a virtual `what` function that returns a description of the error.

`ConnectionFailed` is thrown whenever an error code is reported by the underlying socket API or TLS API. The class contains a public member function `get_is_tls_failure` which indicates whether the error comes from the TLS API or the socket API. An alternative to this could be to have separate error types for TLS errors and TCP/IP socket errors, but having a single error type makes the interface simpler in the most common case where this distinction is not required.

`ResponseParsingFailed` should never be thrown if a request is made to a server that conforms to the HTTP/1.1 specification. However, this type is still provided since there are separate modules for parsing response data. These could still be given invalid input if used separately, so the error type is provided.

5.4.4 Performance

Most of the runtime of an HTTP request is spent sending and receiving data over the network. This makes performance improvements to most of the code irrelevant in practice. However, thought was put into things like minimizing unnecessary copying of data. If a large amount of data is received from the peer, a performance hit may be noticed if it is copied many times, in the parsing algorithms for instance. As mentioned in **5.4.2**, `std::span` is utilized whenever possible and safe to pass around mutable and immutable references to blocks of data instead of potentially copying. Similarly, `std::string_view` is used for immutable references to strings or parts of strings. General best practice rules such as passing small objects by value to avoid unnecessary indirections, marking objects `const` if they are not mutated, and using move semantics (Stroustrup and Sutter, 2020) were followed even if the total performance difference has not been measured and might or might not be significant. Many of these practices benefit both performance and safety, so there was no reason not to follow them.

5.4.5 Safety

Many, if not all of the language and library features discussed in **5.4.2** improve safety and reduce bugs in some way. Old features are also reaching their full potential for safety in newer standards. Being able to use RAII easily with arbitrary C APIs (thanks to lambdas in unevaluated contexts) makes memory leaks and use-after-free errors impossible. I have observed that the ISO C++ standards committee puts a high priority on features that can contribute to the safety and performance of programs. Following new best practices that rely on these features, I have noticed how much less time and effort I have been spending on debugging runtime errors - and how much more time I have been spending on compile-time errors, compared to my previous development. In my view, this is a very positive change. Taking advantage of the compiler to save the developer early is better than searching every corner of a large jungle later on just to find a small use-after-free error that crashed the program.

5.4.6 Testing

Putting effort into writing extensive unit tests is also something that can save the developer early. To make unit testing as effective as possible, all non-trivial functionality that can be tested as a unit/component should be tested. It was found that following the Single Responsibility Principle helped with this. By breaking up functionality into smaller modules that have their own responsibilities and are loosely coupled, they can be individually tested with ease.

The only code that is not unit tested are the `Socket` and `Request` classes, and a few functions that depend directly on these. This is because of three reasons:

1. They are abstractions of APIs which we have no control over, so testing them would not only test the abstraction layer but rather mostly the underlying API.
2. As a consequence of the previous point, writing reliable tests for these components would be next to impossible. The internet is inherently unreliable. A server that we know the exact behavior of would have to be written and a synthetic internet connection would need to be mocked, if not the whole socket API.
3. Sending data over the internet takes time, and unit tests are meant to be quick, especially because of their large quantity.

Instead of doing that, these components were tested in a form of integration tests, which in this case was by writing the example programs located in the `examples` directory and testing them with different servers. The components that cannot be unit tested should be kept as thin as possible, extracting any unit-testable functionality into separate functions and classes.

6 Conclusions

One conclusion that can be drawn is that modern C++ features introduced in recent years provide more capable tools for writing stronger interfaces, safer code, and overall can reduce the time spent debugging runtime errors. However, features are only tools, and they can be used in many ways. Having new, well-designed features does not by itself improve the quality of software. To write Good Code™, especially in such a backward-compatible programming language, a carefully selected set of guidelines is still required and thought needs to be put into software design. The C++ Core Guidelines have shown, at least in this work, to give positive results.

Designing a good abstraction structure for the library was a challenging task, especially because the abstraction structure to a large degree was dependent on which low-level APIs the library was built upon. A conclusion from the development of this work is that it is a good idea, in cross-platform projects built on native APIs, to put thought into choosing the APIs to abstract early on in the process, perhaps by looking for the lowest common denominator among the platforms you are wishing to support. In this case, sockets were the lowest common denominator.

Cpp20InternetClient was admittedly the first project that I wrote unit tests for. Thought was put into separating functionality into modules that have their own responsibilities, with separate, simple interfaces, as this made the modules easier to test separately. It can be concluded that good test coverage can be accomplished by following the Single Responsibility Principle and extracting as much functionality as possible from modules that cannot be unit tested. A good guideline is to keep functions and classes small.

I look forward to seeing how C++20 modules will affect software development in the future, as moving from header files and implementation files to modules will change the way we write code and structure our projects. I am also curious about how coroutines will be utilized in future projects when the standard library has strong support for them.

References

Beck, K. (2002, November 3). *Test-driven development*. Addison-Wesley Educational Publishers Inc.

Bondarenko, K. (2019, May 6). *Static polymorphism in c++*. https://medium.com/@kateolenya/static-polymorphism-in-c-9e1ae27a945b

Catchorg. (2021). *Catch2*. https://github.com/catchorg/Catch2

CppReference. (2020a, November 27). *C++ attribute: Nodiscard (since c++17)*. https://en.cppreference.com/w/cpp/language/attributes/nodiscard

CppReference. (2020b, August 11). *C++ reference*. https://en.cppreference.com/w/Main_Page

CppReference. (2020c, August 11). *Class declaration*. https://en.cppreference.com/w/cpp/language/class

CppReference. (2020d, August 11). *Constrained algorithms (since c++20)*. https://en.cppreference.com/w/cpp/algorithm/ranges

CppReference. (2020e, December 22). *Copy elision*. https://en.cppreference.com/w/cpp/language/copy_elision

CppReference. (2020f, December 30). *Default initialization*. https://en.cppreference.com/w/cpp/language/default_initialization

CppReference. (2020g, September 21). *Fold expression (since c++17)*. https://en.cppreference.com/w/cpp/language/fold

CppReference. (2020h, October 9). *Non-static member functions*. https://en.cppreference.com/w/cpp/language/member_functions

CppReference. (2020i, October 2). *PImpl*. https://en.cppreference.com/w/cpp/language/pimpl

CppReference. (2020j, February 23). *Raii*. https://en.cppreference.com/w/cpp/language/raii

CppReference. (2020k, October 14). *SFINAE*. https://en.cppreference.com/w/cpp/language/sfinae

CppReference. (2021a, February 7). *Constraints and concepts (since c++20)*. https://en.cppreference.com/w/cpp/language/constraints

CppReference. (2021b, January 10). *Coroutines*. https://en.cppreference.com/w/cpp/language/coroutines

CppReference. (2021c, January 14). *Placeholder type specifiers (since c++11)*. https://en.cppreference.com/w/cpp/language/auto

CppReference. (2021d, February 10). *Ranges library (c++20)*. https://en.cppreference.com/w/cpp/ranges

Dionne, L. (2016, August 1). *Lambdas in unevaluated contexts*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0315r1.pdf

Falco, V. (2019). *Boost.beast*. https://www.boost.org/doc/libs/1_75_0/libs/beast/doc/html/index.html

GCC Contributors. (n.d.). *Options to request or suppress warnings*. https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

IEEE. (2018). *The open group base specifications issue 7, 2018 edition*. https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html

IETF. (2018, August). *The transport layer security (tls) protocol version 1.3*. https://tools.ietf.org/html/rfc8446

Jetbrains. (2020). *The State of Developer Ecosystem 2020*. https://www.jetbrains.com/lp/devecosystem-2020/cpp/

Kitware, Inc. and Contributors. (2020). *Cmake reference documentation*. https://cmake.org/cmake/help/v3.18/

Kohlhoff, C. M. (2020). *Boost.asio*. https://www.boost.org/doc/libs/1_75_0/doc/html/boost_asio.html

MDN contributors. (2020, December 30). *Https*. https://developer.mozilla.org/en-US/docs/Glossary/https

MDN contributors. (2021, January 9). *Transfer-Encoding*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding

Microsoft. (2018a, May 31). *About WinINet*. https://docs.microsoft.com/en-us/windows/win32/wininet/about-wininet

Microsoft. (2018b, May 31). *Using SSPI with a Windows Sockets Client*. https://docs.microsoft.com/en-us/windows/win32/secauthn/using-sspi-with-a-windows-sockets-client

Microsoft. (2019a, July 25). *InitializeSecurityContext (Schannel) function*. https://docs.microsoft.com/en-us/windows/win32/secauthn/initializesecuritycontext--schannel

Microsoft. (2019b, July 25). *QueryContextAttributes (Schannel) function*. https://docs.microsoft.com/en-us/windows/win32/secauthn/querycontextattributes--schannel

Microsoft. (2021). *Technical documentation*. https://docs.microsoft.com/en-us/documentation/

ModernesCpp. (2020, May 10). *C++20: The Advantages of Modules*. https://www.modernescpp.com/index.php/cpp20-modules

Munoz, D. (n.d.). *How c++ works: Understanding compilation*. https://www.toptal.com/c-plus-plus/c-plus-plus-understanding-compilation

Nguyen, H. (2017). *C++ Requests: Curl for People, a spiritual port of Python Requests*. https://github.com/whoshuu/cpr

Odzhan. (2017). *Tls.h*. https://github.com/odzhan/shells/blob/master/s6/tls.h#L60

OpenSSL Software Foundation. (2018). *Welcome to openssl!* https://www.openssl.org/

Schreiner, H. (2021, February 11). *An Introduction to Modern CMake*. https://cliutils.gitlab.io/modern-cmake/

StackOverflow. (2021). *Home*. https://stackoverflow.com/

Stenberg, D. (2020). *command line tool and library for transferring data with URLs (since 1998)*. https://curl.se/

Stenberg, D., Hoersken, M., Salisbury, M., & other contributors. (2021). *Schannel.c*. https://github.com/curl/curl/blob/master/lib/vtls/schannel.c#L125

Stroustrup, B., & Sutter, H. (2020, August 3). *C++ Core Guidelines*. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Stroustrup, B. (1995). Why c++ is not just an object-oriented programming language. *SIGPLAN OOPS Mess.*, *6*(4), 1–13. https://doi.org/10.1145/260111.260207

Sundin, B. (2021). *An HTTP(S) client library for C++20*. https://github.com/avocadoboi/cpp20-internet-client

Sutter, H. (2013). GotW #94 Solution: AAA Style (Almost Always Auto). *Sutter's Mill*. https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

The Chromium Authors. (2012). *ssl_client_socket_win.cc*. https://github.com/adobe/chromium/blob/master/net/socket/ssl_client_socket_win.cc#L370

The Internet Society. (1999). *Hypertext Transfer Protocol – HTTP/1.1*. https://tools.ietf.org/html/rfc2616

The University of Glasgow. (n.d.). *Network.socket - sending and receiving data*. http://hackage.haskell.org/package/network-3.1.2.1/docs/Network-Socket.html#g:22

YouTube. (2020, July 23). *Correct by Construction: APIs That Are Easy to Use and Hard to Misuse*. https://www.youtube.com/watch?v=nLSm3Haxz0I

YouTube. (2021a, March 1). *C++ Weekly With Jason Turner*. https://www.youtube.com/user/lefticus1/videos

YouTube. (2021b, February 3). *Cppcon*. https://www.youtube.com/user/CppCon/videos