## Setting Up CMake

Choice of build structure

## Setting Up Gtest/Gmock

As soon as my project was started, I decided it would need unit tests. It would help in tracking down problems now, and it meant I could guarantee that individual methods were working, compared to the whole picture.

After experimenting with building cppunit and Boost.Test, I realised that mocking would be a very useful tool to have, and neither of those projects supported it. I eventually settled on GoogleTest and GoogleMock.

This made testing a great deal easier as it meant that there was a large user base and a strong customer base behind it. It also meant the unit test and mocking capabilities were built “into” each other, and I didn’t have to worry about making sure the test and mocking frameworks worked together comfortably (and uniformly across files).

The GitHub project also provided some CMake targets that automatically downloaded, built, and linked them with my unit tests. Meaning I don’t have to worry about other users having it pre-installed (just whether they an internet connection).

## Project structure

Originally, my project structure was very flat. The top-level directory contained a single CMakeLists.txt and the main.cpp (for ProtDev itself). With each library/component of the program in a folder in there. Each folder then contained its own tests/ folder that held the unit test files. This method became cumbersome and very hard to maintain. So, in response I did more research into the best way of handling a CMake project and I eventually settled on the system I have now [<https://rix0r.nl/blog/2015/08/13/cmake-guide/>]. Instead of every single package needing to be referenced in the top-level CMakeLists.txt Every single library is its own self-contained project that is exported to the larger CMake Project I’m working with. It also allows for more implementation hiding between libraries. And it helps to keep all my executable files much tidier (along with executable targets being a great deal simpler to define).

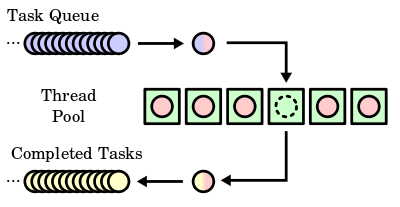
## Moving From Raw Pointers To Special Pointers

A new feature of C++11 and up is the addition of *special pointers*. These special pointers are reference counted (thus are deleted automatically when nothing is referencing them) and allow finer control over how they should behave as “pointers”. When I realised that modern C++ advocates *against* the use of traditional raw pointers, I set about updating what I had written to the current standard.

This was a very difficult task as at the time I had already written a great deal of code and I did not completely understand special pointers, nor their individual uses when I began. However, I feel as though I have a much more robust and well managed piece of software now that it’s more conforming to modern standards.

## Generic Thread Pool

The most significant feature of ProtDev is its ability to handle multiple threads sending data to a target. For increased control and higher bandwidth generally. My original plan was to have threads handled on an as-needed. Whatever object required them would just be handed one. After discussing this method with others, I was made aware of the thread-pool-queue pattern.



1: Picture from Wikipedia (<https://upload.wikimedia.org/wikipedia/commons/thumb/0/0c/Thread_pool.svg/600px-Thread_pool.svg.png>)

This method allowed me to limit the number of threads I had in a more standard manner. However, I would still need a thread pool for every possible return type a thread may have. One method would be to just do that, and create a new handler for each return type, another would be to do away with type safety and use a void\* return type.

C++11 has been a major stepping stone in how to use C++ and bringing it more up-to-date. This includes the “auto” specifier and other functions dedicated to more generic programming that tie in with C++’s <*template*> functionality.

Auto only works for if the compiler can figure out what the type will be upon the instantiation of the variable; however, using the *decltype* specifier from C++11 along with C++14’s ability to deduce a return type depending on template parameters, a totally generic method can be written.

With all the tools available to me in modern standards of C++ I was able to create a single thread pool. Cutting maintenance drastically and creating a good solution that can be used across the system without too much specificity. It’ll also help me in leaps and bounds in managing threads.

## Network Communication

The C++ standard library, though incredibly powerful and featureful, still leaves areas to be desired. Though C++17 is supposed to be addressing these, there’s no proper documentation on it yet (nor is it fully implemented in my compiler, GCC). The basic communication handler will have to be written using the old BSD Sockets library (as they were in C on Linux). This also means that I’ll be forced to use functions (as opposed to methods) which can cause issues with my unit testing framework.

I decided that the absolute first step in this would be to create a class wrapper around the BSD sockets library and pass that in as a parameter to the class. This way I can write a class for mocking outputs/inputs and do things in a more OO way.

To verify results, I am writing and maintaining a simple hellogoodbyeworld program. The idea behind it is that I can run the server and then run ProtDev against it. It’ll run a very simple protocol that simply answers back when spoken to.

## Logger and Testing

The logger has gone through many iterations. The first thing I attempted to implement was the logger service (to try and give me a better idea of how the new C++1x threading features worked and their benefit. The original plan for the logger consisted of having the logs written to using a single thread that sat in a full “log handler”. Each log file object would be passed into whatever component needed it and write to that. The log handler would then loop over each log file and flush its messages to the stream chosen for that log file.

The original logger library was deleted in favour of re-writing it with fresh eyes and more experience into its current state. A great deal of chaff was skimmed out and more significance was placed on using C++ features such as std::condition\_variable; a kind of mutex that allows users to notify other threads with the same variable that they may continue.

## Factories

Due to the goal of the program (plugins), the factory pattern is a major component that will be needed in the required classes. However, their use extends beyond just the plugin system.

I decided that a logger could instead be setup to use more generic streams and repurposed for outputting to *stdin* or *stderr* as well as the file streams they were intended for. This also then required a strategy pattern for choosing how what is sent to the log is outputted. Which then allowed me to package the stream into a std::unique\_ptr that holds a copy of the base class std::ostream.

## Parsing Complications

The agony of parsing…

## Time Keeping/Timer

It was clear from the start that some way of tracking the time would be very important, however due to a want to just continue with the tasks I left the timer until one of the last libraries due to its frustrating complexity and its relative unimportance (as a proof of concept).

The timeTicker class has gone through many several iterations using C++’s own std::chrono library and using timeval/timespec. However, none of them worked as well as I wanted them to. Eventually, after much of the main program was completed I finally implemented the timeTicker class using timeval and string manipulation.

Std::chrono wasn’t used as it works better when the time doesn't’ need to be outputted into any sort of string format (and I have less experience in it) The solution that I have now will be perfectly adequate for my needs and I can work on improving it at a later date.

The TimeTracker library was what I finally came up with and includes two implementations. One for using timeval (seconds, microseconds) and one for using timespec (seconds, nanoseconds). These two possible paths allow for

## API

The API was one of the harder parts of the program as it was using technologies that I am in no-way familiar with. It also required a lot of third-party libraries to make work; both in the API itself and in the communication using Google’s protobuf technology.

The first step was in figuring out the best way to package the required libraries or potentially to just distribute them with the software itself (to make building easier later on).

Protobuf is, unfortunately, too complicated (and in-compatible with cmake) to be able to include. GRPC, however, can be included as a part of the source in the “otherLibraries” directory which, with some tinkering, can handle it’s protobuf dependency too.

The next issue was using protobuf to generate classes based on the protobuf format that will be used to request information through gRPC and then integrating those source files into my own source tree.

After that, it’s a matter of learning how to use

## How to Model Chaos

Currently outstanding