# wee UWS logo

**Computer Games Technology**

Compile-time Introspection in C++

**Computing Honours Project (COMP10034) Interim Report**

Jonathan Livingstone - B00236297

14/12/2016

**Supervisor: Paul Keir**

**Moderator: Malcolm Crowe**

Contents

[1.0 Introduction 3](#_Toc469572101)

[1.1 The Topic 3](#_Toc469572102)

[1.2 The Problem 3](#_Toc469572103)

[1.3 The Project 3](#_Toc469572104)

[1.4 Gantt Chart 4](#_Toc469572105)

[2.0 Literature Review 4](#_Toc469572106)

[2.1 Background/Other Languages 4](#_Toc469572107)

[2.2 C++ tools which provide introspection facilities. 5](#_Toc469572108)

[2.3 Current State of introspection in C++. 6](#_Toc469572109)

[3.0 Current Work 7](#_Toc469572110)

[3.1 Structure 7](#_Toc469572111)

[3.1 Flags 7](#_Toc469572112)

[3.2 Usage 7](#_Toc469572113)

[3.3 Get Number of Members for a struct. 8](#_Toc469572114)

[3.4 Type comparison 9](#_Toc469572115)

[3.5 Convert a type to a string. 10](#_Toc469572116)

[3.6 Print Struct to Console. 11](#_Toc469572117)

[3.7 Get the number of elements in an enum 13](#_Toc469572118)

[3.8 Convert a string to an enum 14](#_Toc469572119)

[3.10 Convert an enum into a string 14](#_Toc469572120)

[4.0 Internal Details 15](#_Toc469572121)

[4.1 Single Executable. 15](#_Toc469572122)

[4.2 Google Test 15](#_Toc469572123)

[4.3 Custom Parser 16](#_Toc469572124)

[6.0 Future Work. 16](#_Toc469572125)

[6.1 Function Introspection. 16](#_Toc469572126)

[6.2 Error Handling. 16](#_Toc469572127)

[6.3 Standard Template Library Support. 16](#_Toc469572128)

[7.0 Conclusion. 17](#_Toc469572129)

[7.0 Demo source code 18](#_Toc469572130)

[7.1 Struct demo code 18](#_Toc469572131)

[7.2 Enum demo code 20](#_Toc469572132)

[8.0 References 21](#_Toc469572133)

# Introduction

## The Topic

While many programming languages provide complex mechanisms in order to introspect the data and functions of the language itself, this is a feature missing from C++. Other popular languages, such as *Java* and *C#*, allow the programmer to view, and even manipulate the data, at runtime. Some newer languages, such as *D* or *Go*, offer introspection at compile-time, meaning that there is no runtime cost to the introspection. However, introspecting data at compile time means that the metaprogramming facilities offered are more limited, so there are benefits and drawbacks to each way.

## The Problem

Because *C++* lacks introspection, it can make a lot of programming just boilerplate, which takes up a lot of time. If the user wishes to print out a class to the console, for example, they will have to manually type in each member, and print out each one uniquely. This is very error prone, as simply adding a new member variable to the class means that the data being printed out is not a complete representation of the class. Using introspection, this problem can be trivially solved.

## The Project

This project aims to allow C++ programmers to view their data in a similar ways to other performance-orientated languages, like *D* or *Rust*. It will parse a C++ file, and generate a *metafile* for it, which is a standard header file to be included. Inside this header file will be information which allows the user to introspect their data structures in rich and complex ways.

While there are a few ways this project could have been completed, I believe an external tool is the best way to solve the problem. If the tool had been built by extending a current open-source compiler, like *Clang* or *GCC*, then the tool would not be able to be widely accepted. People using the tool would be forced to use a specific compile, which is not even standard-conforming. Having the tool as a separate executable, which generates code, means that it can be used with a wide range of compilers across multiple platforms.

Another way to develop the tool would have been using the GNU Compiler Collection, or the Low-Level Virtual Machine, *LLVM*, to create the external tool. These could have handled the parsing of the C++ language, as well as the standard-conforming code generation. The reason they were not picked was for *speed-of-iteration*. It would have taken a lot of time to set up *LLVM* to work on Windows and Linux, and it would have made the executable harder to distribute because it would require *LLVM* to work. However, as the project grows more and more complex, the parser may use *LLVM* in order to completely support the C++ language.

All of the generated code conforms strictly to the C++11 standard. For older compilers, there are versions of the generated code which conform to the C++98 standard. It has been tested with; GCC version 4.8.4; Clang versions 3.4, 3.5, and 3.8, and Visual Studio versions 9, 10, 12, and 14. It has been tested on Windows 8, Windows 10, and Ubuntu 14.04.5.

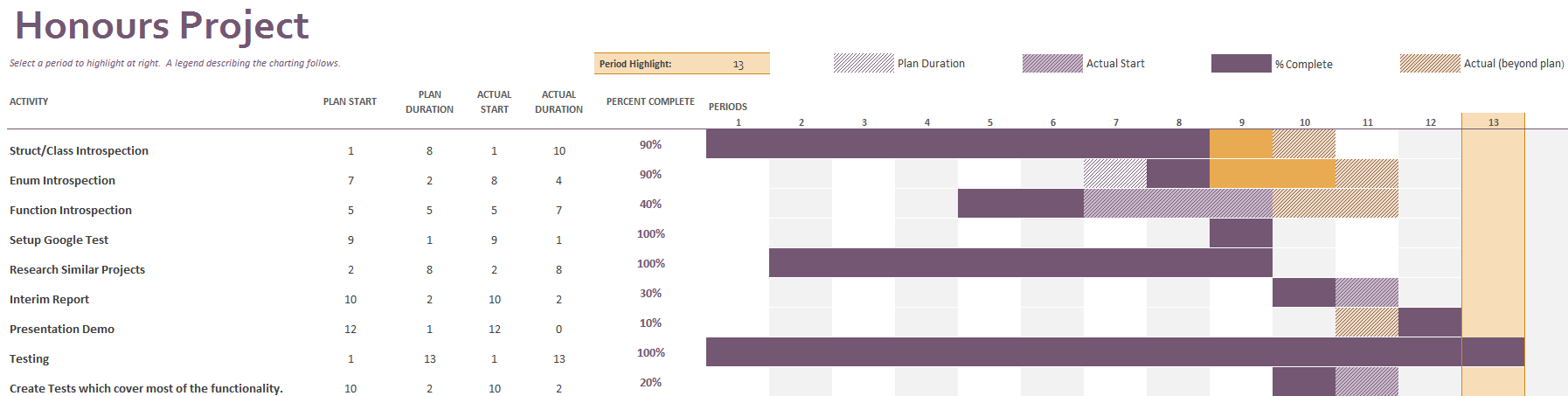
For the rest of the document, when referring to C++ code, I will use the term *struct* to describe a data structure. Because C++ treats *structs* and *classes* the same, except in *classes* everything is *private* by default*,* and the reader may substitute the word *class* in place of *struct*, if they wish.

# Management

## Version Control

The project used *Git* as its primary version control system, storing the code on the website *GitHub*. The reason for choosing *Git* was because it is free, unlike *Perforce*, and it very easy to set up using a website custom repository, unlike *SVN*. Another reason for choosing *Git* was because it allows you to have a full version of the reposition stored locally on your computer, in the *.git* directory, which meant the project could still be worked on when internet was temporarily unavailable.

## Gantt chart



*Figure 1: Gantt chart of progress.*

## Iterative Development

The project is currently using a very iteration-based development cycle. Having small *sprints* of two weeks has allowed a lot of work to be done on the project, and has managed to keep the scope of the project in check.

The project development has also been very *agile*, and a lot of the planning has been added during development. This development style has meant that the focus of the project is constantly been check to make sure the most important features are being worked on.

# Literature Review

This part of the report will analyze the work done on introspection in other programming languages, current C++ tools which provide introspection, and the current state of introspection in the C++ standard.

## Runtime Reflection in other languages

In some other, higher level programming languages, introspection and reflection are very common features.

The language *C#* has some advanced reflection abilities, as well as simple, yet powerful, ones. In C#, every type in the .*NET* framework has a *GetType()* which simply returns the type it is as a *Type* variable (Lischke, 2016). This variable can be used to create new types, or used to compare types. An example of the C# *GetType* method is shown below, comparing two *ints* then an *int* and a *float*. Its output is shown below it.

|  |
| --- |
|  |
|  |

*Figure* ***???****: An example of C#’s GetType method and its output.*

C# also provides ways to retrieve the *properties* of a class at runtime. It does this by allowing each class, which supports the *GetType* method, to also have a *GetProperties* method. The *GetProperties* method can be iterated through to access each element in a class. An example showing this is below.

|  |
| --- |
|  |
|  |

*Figure* ***???****: C#’s GetProperties method used to get all member variables in a class.*

There is also an *IConvertable* class, which can be inherited from, which allows the user to change types at runtime. This is a very powerful introspection ability, which provides something similar to duck typing but with a statically compiled language’s benefits, namely syntax checking. It is also a good example of something which could not be done if the metaprogramming was done at compile time.

The programming language Java has built in introspection and reflection. Roy (2015) talks about the Java Beans API, which provides a lot of functionality to introspect objects. It allows you serialize objects and output their names and values. The Java beans API allows the user to analyze classes to discover properties, methods, and events. While this functionality is definitely a good thing, it has some drawbacks. *Beans* must have a public no-argument constructor, they must have public *getters* and *setters* for each variable, and they must implement the *Serializable* or the *Externalizable* interfaces. These limitations may force the programmer to have to rewrite existing code in order to leverage the introspection features, which is something the preprocessor tools tries to avoid.

The Go programming language has a lot of facilities for reflection built in. This includes the ability to update variables, apply operations to them, and call their functions, without knowing their value at compile time (Donovan, 2015). It allows this by having a *reflection* package. Inside this package, there are two main types; *Types* and *Values*. *Types* represent the actual type of the variable, and *Values* store what the variable actually stores. Using this, it provides ways to convert types to strings, for outputting. This is very similar to how the preprocessor tool works, as it provides both ways to get the type and the value of a variable and convert them to a string, though it does it at compile-time rather than runtime.

## Introspection in D

The programming language D also provides a lot of tools for compile-time introspection. This allows it to have variable introspection, but avoid the runtime costs, unlike most other languages. However, due to this, it can be slightly more limiting than other languages.

Adam D. Ruppe (2014) discusses a powerful introspection feature; *\_\_traits* function, which can retrieves all the compile-time introspection information about its parameter. Using this, you can get everything in a *struct*, including traits, members, methods, and virtual methods.

Some of the examples below show uses of the *\_\_traits* method to discover introspective information about a struct.

|  |
| --- |
|  |
|  |
|  |

*Figure* ***???****: Some example of compile-time introspection in D.*

The language also has an operator called *typeof*, which you can use to test the type of something. The *typeof* operator can be used to create, but it can also be used to compare types, similar to *decltype* in C++. D’s *typeof* can also be used to compare types that are not just variables, however. In D, type comparisons must be wrapped up in an *is* statement, which tests that the type is semantically correct as well as syntactically. Below is an example where it is used compare whether something is a function or not.

|  |
| --- |
|  |

*Figure* ***???****: Using D’s typeof to test whether something is a function.*

## C++ tools which provide introspection facilities.

Because C++ lacks introspection features, some tools have cropped up which allow people to introspect their data.

The most popular library for C++, not including the Standard Library, is Boost, and it provides some aid for serialization. Ramey (2004) created Boost Serialization, and it allows uses to turn classes into a sequence of bytes, from which the entire state of the class can be re-created. However, some limitations of Boost serialization is it requires some intrusive code in order to set it up. This is in contrast to the program specified in this report, which requires no code to set up. It simply provides some helper functions to the programmer, giving them the ability to implement generic features, like serialization, themselves.

One of the most commonly used C++ introspection tools is the *Meta Object Compile*, which will from now on be referred to as *Moc.* *Moc’s* popularity stems from the fact it is coupled with the popular framework *Qt*. *Moc* has some interesting features. One of them is the abilities to access member variables via a string, using the *setProperty* member function. It also creates a complex signals-and-slots framework, which can send a *signal*, which in turns calls all the functions associated with that action. While I do believe that *Qt’s* Moc is a good tool, it has a lot of bugs in the implementation, and a lot of the code is very error prone, and will mask bugs, with no compile error or runtime assert, and just silently fail. It also drags in a lot of code, including the entire *Qt* framework, and keywords, which the user must understand how they work. It also forces the uses into a very specific style of programming, which I wanted to avoid, as I believe a good API should be granular enough to work with others people code, and not force uses to modify their code to work with the tool.

There are various downsides to Qt’s Moc (Oliver, 2016). It is very tightly coupled to the Qt framework, and would thus be unsuitable for a non-graphical application. Going further, however, it would be unsuitable for an application which wants to use introspection, in order to make more readable, robust and/or performant code; and if the user has a different 3D graphics package, whether it’s another open source one or develop in-house, they would have to find a way to integrate their stuff with Qt.

Qt’s Moc, and Qt itself, also have a lot of outstanding issues. Because Qt is trying to be a very large application, which does everything, it has become very buggy. They have an online bug list which is hundreds of entries long, and some of them are years old. I believe my program has an advantage over Qt because of this, because it is very small and is focused on doing one thing and doing it as well as it can, rather than spreading itself very thin trying to do too much.

The Unreal Game Engine has a built-in system, which it calls *Properties*, which are used to provide limited introspection. This is built into the Unreal Engine, and you can *mark* variables as a property by using a keyword before the variable. This could be a macro called *UCLASS* for classes, *UFUNCTION* for member functions, or *UPROPERTY* for member variables. Using this allows developers to introspect and generate their code in very specific, and power ways. Similar to Qt’s *Moc*, the Unreal Property System is mainly used in order to combine UI design and programming in C++. It allows you to create UI in the Unreal Editor, which then calls into a specific C++ function when an action is applied to it, for instance when a button is clicked.

There are many downsides to the Unreal Property System, however. The main issue is how tightly coupled it is to the Unreal Game Engine. There is no real way to separate the two, and thus if you wanted to use introspection in a non-graphical application, it would not be suitable.

Another issue is it has a lot of Unreal-specific keywords it introduced, in the form of macros. Having a lot of these through code can make the code much more difficult to read, as anyone reading it now has to have an understanding of what the Unreal Property System is, how to use it, and what each of the keywords mean. This extra knowledge will make maintaining code, as well just reading others people’s code, much more difficult.

## Current State of introspection in C++.

Chochlik and Naumann (2016) discuss the rational and evolution of static reflection for C++ in their proposal to add it to the language. They discuss adding introspection to C++ so programmers could access features like; the name of a class, its base class, its data members, and any nested information within the class. They also discuss adding a new keyword to C++, *reflexpr*, which is used for the compile-time introspection.

In their paper, they propose introducing *Meta-Objects*, which are created via the *reflexpr.* Their proposal discusses creating constant *structs* for the program to use, but which the compiler fills out at compile time.

The operator they discuss, *reflexpr*, will return a *metatype* to the user conforming to the particular type passed in. This is because the details someone would want from a *struct* are very different than what they would want from a *function*, or an *enum*.

They also discuss possible difficulties. *Unions* would be very difficult to introspect, at least to the limit of *structs*, because of how limited they are in C++. Chochlik and Naumann (2016) discuss whether *unions* should generate their own *metatype* or whether their data should be bundled together in the same type as a *struct*.

They also discuss the difficulties of adding a new keyword into C++, *reflexpr*, which could cause naming conflicts in codebases. However they believe this to be a small problem. They did a scan of 994 open-source repositories on *GitHub* and found no occurrences of “*reflexpr*”.

Below is a small example, using Chochlík’s (2016) fork of clang, where he implemented a version of the proposed reflection facilities.

|  |
| --- |
|  |

*Figure* ***???****: Example showing Chochlík’s clang fork to get the number of members in a struct.*

# Current Work

## The tool

The introspection tool being discussed in this document aims to add compile-time introspection into C++. It has a few design goals, which differ from some other introspection tools.

It is designed to be as non-intrusive as possible. The generated code is very lightweight, and the API assumed very little about the code it’s working with. While some other introspection tools require the user to inherit from special *base-classes* and mark-up their structs, this tools aims to be compatible with vanilla C++ code.

The introspection from within the tool is also designed to run fast. Because the tool tries to provide introspection features for each struct, the *worstTime(n)* will be linear for each struct in the file. However, each function in the tool has been tested and, with the exception of *pp::print* and *pp::serialize*, each function is incredibly fast and should not be a bottle neck.

As well as lightweight, the tool is designed to be backwards compatible with earlier versions of C++. With the exception of some functions, like *print* and *serialize*, the code generated is backwards compatible with the C++98 specification. The functions *print* and *serialize* require a C++11 compiler, but they also have backwards compatible equivalents, for people working on older compilers.

The introspection tool is all contained within one executable file. It does not link to any external *.dlls*, and statically links to the *C Runtime Library*. One of the design goals for the tool was ease-of-use for users of the tool. Hence, it only requires the single executable, and does not require any external files or installers to work.

## Usage

The pre-processor is just a small command-line argument. It is just 226 KB large, and runs roughly as fast as a modern C++ compiler.

If you build from the command line, a simple example of using the *preprocessor* would be:

|  |
| --- |
|  |

*Figure 2: Example using the preprocessor tool and GCC.*

The first line, *preprocessor* *test\_code.cpp*, calls the tool on a sample program. This will generate two files, *static\_generated.h*, and, *test\_code\_generated.h*. The first file, *static\_generated.h*, is a *static* file, which is always written out the same when the preprocessor is run. It has a lot of utility code shared between different generated files. The second file contains all the information required to introspect the C++ data structures.

The second line, *g++ test\_code.cpp*, will compile the file, *test\_code.cpp*. Inside the file *test\_code.cpp* it is assumed to have included *test\_code\_generated.h*. Using the data written into *test­\_code\_generated.h*, the user will be able to simulate advanced introspection of C++ data as if it were built into the language.

Some of the features the user will be able to leverage include:

* Automatic printing of *struct* data, either to the console or into a *char array* buffer.
* Methods which allow the user to loop over members of a *struct*, and get the number of members for a *struct*.
* An ability to convert a *struct* name into a string literal, for debug outputting.
* A simple way to find out how many elements are in an *enum*. All of the *enum* functionality should work with classic C-style *enums*, and more modern C++ *enum* classes.
* The ability to convert a string into the index an *enum* represents.
* The ability to convert an *enum* element into the string-literal version.

## Flags

When calling the program, there are a number of flags the user can pass in. A few of these are only available in debug-builds.

If the user passes the flag *–e* in, then the tool will output errors to the console.

If the user passes the flag –h in, or doesn’t pass anything in, then a help section will be displayed, as well as information how to use it.

In debug builds, there are a few extra flags. These were added to make debugging easier for the developer.

The flag *–s* stands for *Silent*, and means that no code will be generated. This was useful for testing, because often it was useful to see if the tool could successfully parse a piece of code or not, but without caring about the output.

The flag *–t* is used to run tests. The tests are run through the Google Test framework, which is only linked in debug builds. It will then run all the tests on the tool and check that it’s okay. Most of the tests that run through Google Test make sure that the parser can handle difficult syntax. Passing *–t* in a debug build will only run the tests in a 64-bit build. This is, because of the 2 GB memory limitations of 32-bit builds on Windows, Google Test often ran out of memory during testing.

|  |
| --- |
|  |

*Figure* ***???****: Example calling preprocessor and passing flags in.*

## Get Number of Members for a struct.

|  |
| --- |
|  |

*Figure 3: Function definition for pp::get\_num\_of\_members.*

|  |
| --- |
|  |

*Figure 4: Example using pp::get\_num\_of\_members.*

The example above shows one of the most basic levels of introspection offered by the tool. The part which says *pp::get\_num\_of\_members(V2)* simply returns an unsigned integer value which has the number of member variables in the struct. This is useful as the number of elements in the *struct* may change, yet this function will always return the same result.

## Type comparison

Because C++ was not designed with introspection in mind, there are some design choices which can make it difficult to implement.

An example of this that C++ forbids the comparison of types. The following line will not compile under any standard-compliant C++ compiler.

|  |
| --- |
|  |

*Figure 5: Invalid type comparison.*

While that may seem like a trivial example, it has far reaching consequences. The C++11 keyword, *decltype*, for example, is much more limited because of this. For example, the follow code will not compile.

|  |
| --- |
|  |

*Figure 6: Invalid type comparison using decltype.*

This can also have a negative effect on templated code, as the following code will not work either.

|  |
| --- |
|  |

*Figure 7: Invalid type comparison in template code.*

The metaprogramming tool, however, exposes three mechanism for comparing types.

|  |
| --- |
|  |
|  |
|  |

*Figure 8: pp::type\_compare, pp::fuzzy\_type\_compare, and pp::weak\_type\_compare functions.*

Each of these functions does something slightly different. *pp::type\_compare* does a vanilla comparison on the types. *pp::fuzzy\_type\_compare* test whether the *structs* are the same, or if one is a *base* *struct* of another. *pp::weak\_type\_compare* tests whether two types are the same, ignoring whether one is a pointer or not.

Using this, it is possible to make all of the previous examples work. All of the following examples of code will compile, when the program has used the tool.

|  |
| --- |
|  |

*Figure 9: Basic pp::type\_compare example.*

|  |
| --- |
|  |

*Figure 10: pp::type\_compare example using decltype.*

|  |
| --- |
|  |

*Figure 11: pp::type\_compare example in template code.*

The function will only return *true* on exact type comparisons. For similar types, like *int* and *long*, or related types, like *class* and its *subclass*, it will still return *false*. The following code excerpt shows examples of different type comparisons, and the result they would yield.

|  |
| --- |
|  |

*Figure 12: List of pp::type\_compare examples.*

## Convert a type to a string.

|  |
| --- |
|  |
|  |

*Figure 13: pp::type\_to\_string and pp::weak\_type\_to\_string definition.*

This function will convert a *struct* type into a *string literal*. The first function, *pp::type\_to\_string,* will convert into a string of its exact type. The second, *pp::weak\_type\_to\_string,* will convert into its base type, regardless of whether it’s a pointer or not. Below is a simple example of how they it could be used, and their output.

|  |
| --- |
|  |
|  |

*Figure 14: pp::type\_to\_string example and its output.*

A good use for this could be debugging templated code. C++ templates can be very difficult to debug, and they often produce very hard-to-decipher error messages. This can get worse with nested template code. The following code except shows how this function could be used in a simple template in order to print the type of a variable and its value.

|  |
| --- |
|  |

*Figure 15: pp::type\_to\_string in function.*

## Get base type.

|  |
| --- |
|  |

*Figure* ***???****: pp::get\_base\_type\_count definition.*

Figure ??? above show the definition for a function which will return how many *structs* the *struct* passed in inherits from. An example of using it is shown below in figure ???.

|  |
| --- |
|  |
|  |

*Figure ???: pp::get\_base\_type\_count example and output.*

## Print Struct to Console.

One of the most powerful methods available inside the preprocessor is used for printing a *struct* to the console. The function definition is provided below, as well as a simple example using it, and the output of the example.

|  |
| --- |
|  |

*Figure 16: pp::print function definition.*

|  |
| --- |
|  |
|  |

*Figure 17: pp::print function example.*

The default parameters on the function *pp::print* are used so the programmer can use their own buffer to print the memory. If no parameters are set, then the function must allocate some memory to print the value. An example, which has the same output, but avoid the memory allocation is shown below.

|  |
| --- |
|  |

*Figure 18: pp::print function using custom buffer.*

In the future, a function similar to this could be used to serialize data structures into different formats, such as *XML* or *JSON*, without the programmer having to do any work on their part.

A related method, which just serializes a *struct* into a buffer without printing it, is also available. Its definition is shown below.

|  |
| --- |
|  |

*Figure 19: pp::serialize function definition.*

The *pp::print* and *pp::serialize* methods have been tested very thoroughly. They have some limitations, however, which are that they must adhere to the C++11 standard definition of a *Standard Layout Class*. This means that it must have:

* No static data types,
* No virtual functions and no virtual base classes,
* All of the members must be *standard layout classes*.
* All of the base classes must be *standard layout classes*.

It should be noted that they may still work if some of these conditions are not met, but the result will be implementation-specific, depending on the compiler.

Both *pp::print* and *pp::serialize* functions require a C++11 compiler, as they use the *decltype* operator. There are alternate versions available, however. These are *pp::print\_type­* and *pp::serialize\_type* and their function definitions are shown below.

**Add function defintions.**

## Get the number of elements in an enum

In C++, there is no way to get the number of elements in an *enum*. Many developers attempt to overcome this by adding a value at the end of the *enum*, usually called *COUNT*, which will give the number of elements in the *enum*. The following code except is an example of this.

|  |
| --- |
|  |

*Figure 20: Common C++ idiom to get number of element in an enum.*

While this does go partway to solving the problem, it has some obvious limitations. If another developer, who is unfamiliar with this paradigm, modifies the *enum* and puts an index after *COUNT*, then *COUNT* will be wrong. Also, in modern C++, it is possible to assign specific values to *enum* elements. If the index C was assigned to 10, for instance, then *COUNT* would be 11, rather than 3.

Using the preprocessor tool, we can get around this. It provides a function, *pp::get\_number\_of\_enum\_elements*, which allows you to directly get the number of elements in an *enum*, regardless of the order or elements or what their values were assigned to. Its prototype is in the code excerpt following.

|  |
| --- |
|  |

*Figure 21: pp::get\_number\_of\_enum\_element function definition.*

It can be called just like a normal function.

|  |
| --- |
|  |

*Figure 22: Example using pp::get\_number\_of\_enum\_elements function.*

I believe this provides a much more robust way to get the number of elements in an *enum*, and is much more fault-tolerant than the previous way mentioned.

## Convert a string to an enum

Similar to the programming language C#, the metaprogramming tool provides a way to convert a string into an *enum* element. The function has the following signature.

|  |
| --- |
|  |

*Figure 23: pp::string\_to\_enum function.*

The first parameter is the type of the *enum*, and the second is a string which should match the *enum* element. Following is an example of using the method.

|  |
| --- |
|  |

*Figure 24: pp::string\_to\_enum example.*

## 3.10 Convert an enum into a string

Going the opposite way from the previous example, you can also convert an *enum* element into a string literal. The function definition for doing that is following.

|  |
| --- |
|  |

*Figure 25: enum\_to\_string function definition.*

|  |
| --- |
|  |

*Figure 26: pp::enum\_to\_string example.*

In the previous example, the user converts the *enum* index *one* into a string literal containing “*one*”. I believe this would be useful for outputting *enums* in a human-readable way.

# Internal Details

## Single Executable.

The preprocessor is just one executable, *preprocessor.exe*. It does not require any *.dlls* to run. It statically links the C-Runtime Library, so it does not have to be distributed with the *dlls* for that either. This was done because, on Linux shared libraries generally work well, on Windows they do not. Most applications must ship with their own version of the *CRT*, and if that gets updated it may possibly break applications that use it. It also allows the user to have multiple versions of the program on their machine without any conflicts.

## Google Test

The Google Test framework was used in order to test the parser, and find bugs quickly. It allowed large restructuring of the codebase to take place, while ensuring existing functionality kept working.

The following code is a simple example of a test, which makes sure the number of members in a *struct* is correct.

|  |
| --- |
|  |

*Figure 27: Source code demonstrating use of Google Test.*

First, the code creates a dummy string, which has a simple *struct* with 3 members. Then, it passes this string into the *parse\_struct* function, which returns a *StructData* data structure containing all the relevant information on the *struct* parsed. Finally, it does a simple comparison to make sure the number of members parsed is actually 3. If the number of members was not 3, then an assertion would fire when the code was built, and it would output the message and which test failed.

## Custom Parser

The project uses a custom C++ parser, rather than a current open-source one, because of the limited choices available. None of the parsers - GCC\_XML or ANTLR4, - support C++ templates. Because of these limitations, and because of the limited parts of C++ that the tool actually has to parse - it only needs to parse class definitions and function prototypes - it was more expedient to write a custom one rather than use a pre-existing one. However, the parser is very segregated from the other parts of the program, so if a strong C++ parser was found, it would not be much work to switch it in.

# Future Work.

## Function Introspection.

Right now, there is no function introspection data generated. The parser does currently handle functions, and stores data on them, but they are not written out to disk for the user to have access to. While this would not be much work to add, the uses-cases for function introspection are much weaker than for *struct* or *enum* introspection.

## Error Handling.

Right now, a syntax error in normal code may generate a syntax error in the generate code. And, because the generate code appears before the normal code in the compilation unit, it may appear that the generated code is the problem. One of the things I do to combat this is, at a basic level, is to look out for such issues when generating code, then output errors for the user to read. These errors could either be directly printed to the console, or written to *stderr*.

## Standard Template Library Support.

Currently, the preprocessor does not support any of the C++ Standard Template Library. However, because it is a core part of C++, I will support it in the future. This will be possible because all of the containers in the Standard Template Library are well documented, so I will add code which specifically handles them.

Without adding some form of in-code annotations, however, it would be impossible to support custom containers in the preprocessor. As such, there are no plans to support them.

# Conclusion.

I believe I have made good progress on this project. Development of the project was very agile in its approach, usually based around 2-week iterations. By focusing on finishing features within the 2 week period, and trying to implement the features as close to completion as possible, it allowed me to get a lot of work done. More than that, it also meant that I had confidence that the features I had implemented were implemented well and would not break further into development.

During the development, I feel I have gained a greater understanding about introspection, and have a good idea what a lot of the issues putting it into a language.

One of the simpler issues is having a clean interface to gain this introspected information. If the programmer has to go through a lot of difficult-to-read code, rather than simple implementing something the non-introspection way, it weakens the argument for introspection and metaprogramming. This is despite the benefits that introspection can bring to code robustness and future-proofing.

Another issue is getting this data. Because of the way the C++ language parses, which it largely inherited from C, even just adding introspection into the language can prove difficult. Other languages, like D, don’t depend on the order of compilation, and have a module system for including files, which mean the introspection data is gather *before* the program has even begun properly parsing. In C++, however, the language is parsed from the top down. Because of this, it can lead to some difficult problems when generating introspection data for a *struct*. An example would be, if a *struct* has another *struct* as a member pointer, but the second is only forward declared, not properly defined, then the compiler wouldn’t necessarily have the information on-hand generate introspection data. This would mean another compiler pass would be necessary to deal with these situations, which would increase compile times. One of the benefits of having the preprocessor as an external tool, which is *not* built-in to the compiler, means this data can be parsed and generated before the compiler has to do anything, meaning it doesn’t add an significant time to the code generation process.

# Demo source code

## 

## Struct demo code

|  |
| --- |
|  |
|  |

*Figure 28: Advanced struct introspection example.*

## Enum demo code

|  |
| --- |
|  |
|  |

*Figure 29: Advanced enum introspection example.*

# References

Becker P. (2011). *Working Draft, Standard for Programming Language C++.* Available at: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>

Clang. (2016). Available at: <http://clang.llvm.org/>

GCC. (2016). Available at: <https://gcc.gnu.org/>

Git (2016). Available at: <https://git-scm.com/>

Github (2016). Available at: <https://github.com/>

Microsoft. (2016). *Visual Studio*. Available at: <https://www.visualstudio.com/>

Chochlik M., Naumann A. (2016). *Static Reflection; Rationale, design and evolution. 4th Rev.* Available from: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0194r0.pdf>

## Chochlik M. (2016). Mirror reflection utilities. Available from: <https://github.com/matus-chochlik/mirror>

Donovan A., Kernighan B. (2015). *The Go Programming Language*. 1st ed. Available from: <https://www.amazon.co.uk/Programming-Language-Addison-Wesley-Professional-Computing-ebook/dp/B0184N7WWS/ref=sr_1_1?s=books&ie=UTF8&qid=1481752148&sr=1-1&keywords=The+Go+Programming+Language>

GCC XML. (2015). *GCC\_XML.* Available at:[*http://gccxml.github.io/HTML/Index.html*](http://gccxml.github.io/HTML/Index.html)

Lischke M. (2016). *ANTLR4.* Available at: <http://www.soft-gems.net/index.php>

Oliver G. (2016). *Qt’s Moc*.

Roy, U. (2015). *Advanced Java Programming*. 1st ed. Available from: <https://www.amazon.co.uk/Advanced-Java-Programming-Uttam-Roy/dp/0199455503/ref=sr_1_1?s=books&ie=UTF8&qid=1481752130&sr=1-1&keywords=advanced+java+programming>

Ruppe A. (2014). *D Cookbook*. 1st ed. Available from: <https://www.amazon.co.uk/D-Cookbook-Adam-D-Ruppe-ebook/dp/B00KLAJ62M/ref=sr_1_1?s=books&ie=UTF8&qid=1481752218&sr=1-1&keywords=d+cookbook+programming>

Ramey R. (2004). *Boost Serialization.* Available at: <http://www.boost.org/doc/libs/1_62_0/libs/serialization/doc/index.html>

Unreal Games. (2016). *Unreal Engine Property System.*