# B00236297 – Honours Interim Report.

# Introduction

Reasons.

* If I developed built-in introspection by adding to a current open-source compiler (like Clang or GCC), then people couldn’t use it except for testing. Having an external app, which generate standard-compliant C++, means it can be used with most compilers. The generated code has been tested in GCC, Clang, and MSVC.
* I wrote my own parser, rather than using one, because there wasn’t much choice for using an external parser. I had a look at a lot of C++ parser, but the only one that seems useful was GCC (something). But the only parsing issue I expected to run into was templates, and that parser did not properly support templates. So I felt there was no reason to use it.

# 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 C++.

## Background/Other Languages

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

Some languages off Run-Time reflection, which can occur significant performance penalties for using reflection. These languages allow the programmer to decide to make the trade-off if the reflection is worth any performance overhead.

<https://msdn.microsoft.com/en-gb/library/mt656691.aspx>

The language C# has some advanced reflection abilities. In C#, every type, whether it’s a primitive, class, or struct, has a built-in method called *GetType()* which simply returns the type it in. 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 languages benefits, namely syntax checking. It is also a good example of something which could not be done in a compile time

<http://docs.oracle.com/javase/tutorial/reflect/index.html>

<http://web.archive.org/web/20090226224821/http://java.sun.com/docs/books/tutorial/javabeans/introspection/index.html>

The programming language Java has built in introspection and reflection. The Java Beans API provides a lot of functionality to introspect objects. It allows you serialize objects and output their names and values. As well as an introspection API, Java also has a reflection API. This can be used to get the type of an object, and create other objects of the same type at run time.

Other languages, usually more modern ones, saw the runtime overhead of introspection and instead implemented compile-time introspection. This has the benefit of having zero overhead, yet it can be more limited than doing it at runtime.

<http://merbist.com/2011/06/27/golang-reflection-exampl/>

Matt Aimonetti discusses how the programming language Go has introspection, which can be used to loop through its members. This is a very useful feature, yet something C++ is completely without. Using the introspection system I’ve developed, this should be possible in C++ without modifying the core language.

<http://dlang.org/spec/traits.html#allMembers>

The programming language D also provides a lot of tools for compile-time introspection using the *\_\_traits(allMembers)*. Using this, you can get everything in a struct, including traits, members, methods, and virtual methods. The language also has an operator called *typeof*, which you can use to test the type of something. An example would be *is(typeof(member) == function))* to test if something is a function or not.

## C++ tools which provide introspection facilities.

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

<http://www.boost.org/doc/libs/1_61_0/libs/serialization/doc/index.html>

The most popular library for C++, not including the Standard Library, is Boost, and it provides some aid for serialization. Boost serialization 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 my system, *preprocessor.exe*, which requires no code to set up, and simple provides some helper “functions” to the user in order to implement features like this themselves.

<https://woboq.com/blog/reflection-in-cpp-and-qt-moc.html>

Arguable the most commonly used C++ introspection tool is *Moc*, because 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 now 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.

Oliver Offart, the primary developer for Qt’s Moc, discusses what would have to be added to C++ in order to remove the tool Moc completely. His list is quite long, and I’ll fill it in later.

## Current State of introspection in C++.

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0385r0.pdf>

Matus Chochlik and Axel Naumann 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.

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0194r1.html>

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4452.pdf>

# Program

## Structure.

The introspection tool, *preprocessor.exe*, is all contained with the one file. It does not link to any external *.dlls*, and static 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.

## 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 with 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 fun 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 run through Google Test simple 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 2Gb memory limitations of 32-bit builds on Windows, Google Test often ran out of memory during testing.

## Usage.

The pre-processor is just a small command-line argument. It is just 226Kb 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:



The first line, *preprocessor* *test\_code.cpp*, calls the pre-processor 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

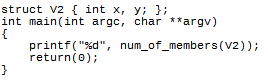
The second line, *g++ test\_code.cpp*, will compile the file, *test\_code.cpp*. Inside the file *test\_code.cpp* it is assumed to have included *test\_code\_generated.h*. Using the data written into *test­\_code\_generated.h*, the user will be able to simulate advanced introspection of C++ data as if 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. This feature will even work with C++11’s *decltype* operator.
* 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 a enum represents.
* The ability to convert an enum element into the string-literal version. This will work indirectly as well, like if a variable has been assigned the value and is passed in, rather than the index directly.

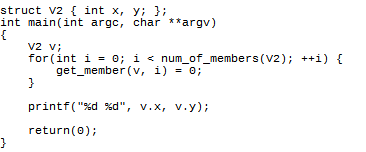
# Struct Introspection.

## Get Number of Members for a struct.

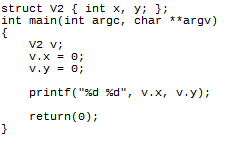
One of the first features I started implemented was introspection of struct data. C++, while a powerful language, almost completely lacks and introspection. A large consensus of people online believe this is because the generated C++ code would encounter a performance hit, however I actually believe this is incorrect. When the compile is parsing code, it is building a syntax tree which could simple be available to the programmer.



Looking at the example above, if the C++ standards committee had added an operator *num\_of\_members* which simple returned an integer of how many members were in a struct, it would not occur any runtime overhead, as *num\_of\_members(V2)* would simple be replaced by the number 2 at compile time, the same was *sizeof* works.



Building upon the previous example, and adding a new operator *get\_member*, which simply returns the member at index *i*. the follow code could also be compiled without any runtime overhead. Because the compiler knows how many members are in the struct *V2*, and the C++ standard and that an optimizing compiler is not allowed to rearrange members, the compile has all the information to make the code work. It would be very similar to the following code, except more robust to changes.

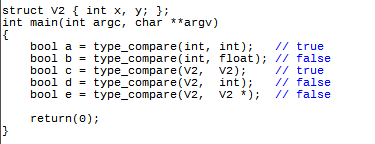


The compile-time introspection system illustrated in this document attempt to recreate some of these features. While they will never be as good as if the C++ standard committee had simply added them into the language spec, I believe they provide a very good alternative. The generate code is compatible with most versions of C++.

## Get the type of a member variable in a struct.

## Compare two variables to see if they’re the same.

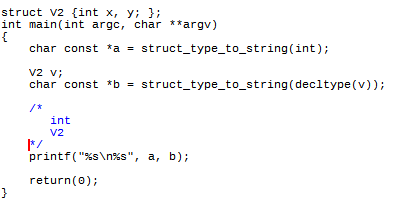
This feature is actually just some template trickery, rather than generated like most of the others.



In C++, you cannot compare types, so the code *if(int == int)* will not compile. While this may seem like a silly example, if you include C++11 code, and the keyword *decltype* then it becomes more obvious why you would want this.

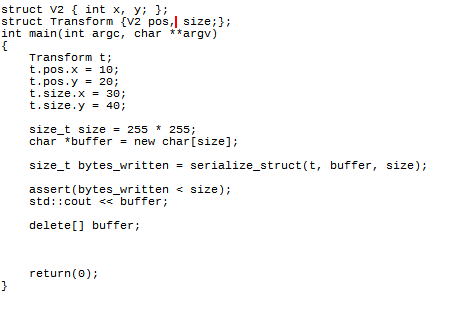
## Convert a type to a string.

This function takes a primitive or struct type as its first argument, and will convert them into a string literal.

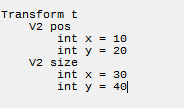


In the above code, you can see some of the debug uses for this. The above code will also work with C++11’s *decltype* keyword.

## Print Struct to Console.



The above code example produces the following output (taken from the Visual Studio Watch Window).



As you can see, it has correctly serialized the struct. The point of serializing the struct was not for sending across data packets, or writing to file, but for outputting in a graphical display. The main use case for this would be, in a computer game, if you ran a debug build of the game, you could serialize all the object in the game when the developer clicks on them. This would provide real-time debugging information to the developer.

The serialize struct method has been thoroughly tested and works with multiple inheritance, pointers, and normal data types.

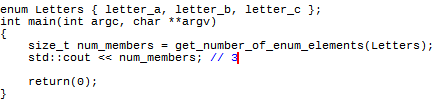
## Print Struct To Buffer.

# Enum Introspection.

All the enum code has been tested with C’s enums, and C++ enum class. It works with both.

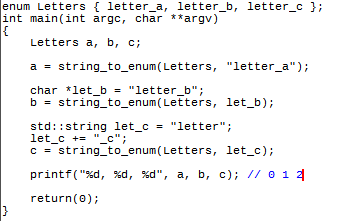
## Get the number of elements in an enum.

This code will output the number of elements in an enum.



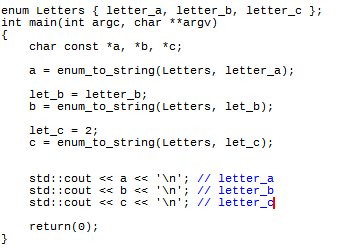
A lot of people, when writing enums, will have a count element on the end, so they can always get the number of elements (in the previous example, after *letter\_c*, there could be an element called *letter\_count*). This has some disadvantages of, including; someone may accidentally move count so it’s no longer at the start, or add elements after it without properly reading how the enum works. Or, if the user assigns a number to one of the elements in the enum, like *letter\_c = 5*, then the element *letter\_count* would not be wrong. Using the *get\_number\_of\_enum\_elements* “function”, the user now has a more robust way to find out how many elements are in an enum.

## Convert a string to an enum.



This is used to convert a string into an enum. From the above example, you can see how this would be useful if you were reading data from a file.

## Convert an enum to a string.



This method takes an enum index and will convert it into a string. This would be useful for outputting enums in a human-readable way. It would also be useful for serializing enums.

# Internal Details

## Single Executable.

The preprocessor is just one executable, *preprocessor.exe*. It does not require any *.dlls* to run. It static links the C-Runtime Library, so it does not have to ship with that either. My logic for that was, it can be a pain to use installers, and it can be easy to mess up *.dlls*. Having just the *.exe* means that the user only has one program, and it also means it is easy for the user to switch machine with it. It also means there can, very easy, be two versions of the preprocessor on one machine.

## Google Test.

I used the google test in order to test a lot of the parser, and find bugs quickly. It also allowed me to make large changes to the codebase and still be sure everything worked.

# Future Work.

## Function Introspection.

Right now, there is no function introspection data generated. While this would not be much work to add, the uses-cases for function introspection seems much weaker than for struct and enum introspection.

## Error Handling.

Right now, a syntax error in your code may generate a syntax error in the generate code. And, because the generate code appears before your code in the compilation unit, it may appear that the generated code is the issue. One of the things I do to combat this, 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 directory printed to the console, or written to *stderr*.

## Standard Template Library Support.

Currently, the preprocessor doesn’t have any of the standard template library. However, because it is a core part of C++, I will support it in the future.

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

# Conclusion.