# B00236297 – Honours Interim Report.

# Introduction

Reasons.

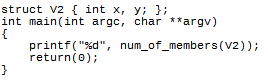
* If I developed built-in introspection by adding to a current open-source compiler (like Clang or GCC), then people couldn’t use it except for testing. Having an external app, which generate standard-compliant C++, means it can be used with most compilers. The generated has been tested in GCC, Clang, 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

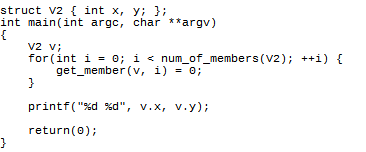
# Struct Introspection.

## Get Number of Members for a struct.

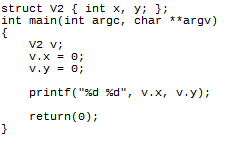
One of the first features I started implemented was introspection of struct data. C++, while a powerful language, almost completely lacks and introspection. A large consensus of people online believe this is because the generated C++ code would encounter a performance hit, however I actually believe 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.

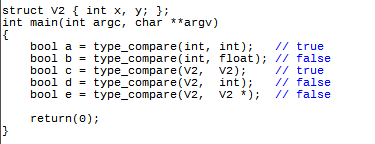


The compile-time introspection system illustrated in this document attempt to recreate some of these features. While they will never be as good as if the C++ standard committee had simply added them into the language spec, I believe they provide a very good alternative. The generate code is compatible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

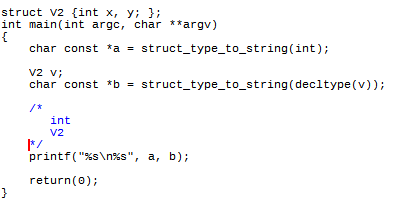
This feature is actually just some template trickery, rather than generated like most of the others.



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.

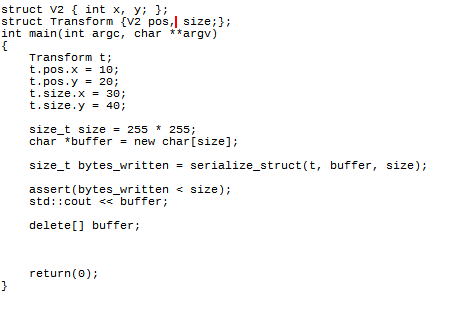
## Convert a type to a string.

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

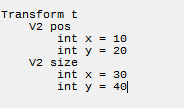


In the above code, you can see some of the debug uses for this. The above code will also work with C++11’s *decltype* keyword.

## Serialize a struct for output.



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.

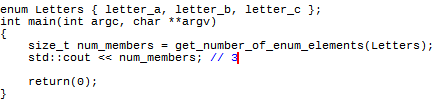
The serialize struct method has been thoroughly tested and works with multiple inheritance, pointers, and normal data types.

# Enum Introspection.

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

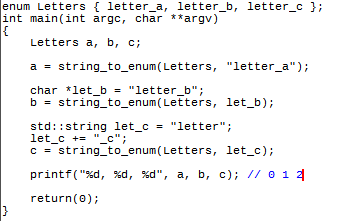
## Get the number of elements in an enum.

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



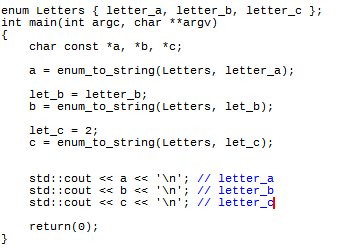
A lot of people, when writing enums, will have a count element on the end, so they can always get the number of elements (in the previous example, after *letter\_c*, there could be an element called *letter\_count*). This has some disadvantages of, including; someone may accidentally move count so it’s 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. I also allowed me to make large changes to the codebase, and still be sure everything worked.

# Conclusion.