

**Computer Games Technology**

Introspection in C++

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

Jonathan Livingstone - B00236297

??/??/2017

**Supervisor: Paul Keir**

**Moderator: Malcolm Crowe**

## Abstract

The C++ programming language, until very recently, almost completely lacks any way to introspect its data types and functions from within the language itself. The C++11 specification [] goes some way to allowing this, with the use of *type\_traits* **REFERENCE**, and C++ goes even further with the use of *structure* bindings **REFERENCE**. Despite this, however, they do now allow much more than toy examples and simple introspection of data. While there are some 3rd party libraries which aid this, these often come with a lot of negatives and can be very complicated to work with, often requiring the programmer to rewrite their code how the library wants it. This piece discusses a tool which aims to provide introspection in C++ in a robust and easy-to-use way. This is so C++ programmers can write more robust code using introspection features common in other languages, without making performance tradeoffs to get them. The main design goal of this tool is that it should be very easy to use. It should be able to work with most C++ project with minimal work, and be fast enough so it’s not a burden on build times. It should also be easy to work with from within the code, and not have a lot of complicated frameworks and implicit knowledge for the user to understand in order to use the tool.

## 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 compiler, 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, and should work on Windows, OS X, or Linux operating systems under either MSVC, GCC, or Clang compilers. Some of the examples presented in this document, however, require C++14 or C++17 compilers to run, but this has been noted.

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

# Management

## Software used

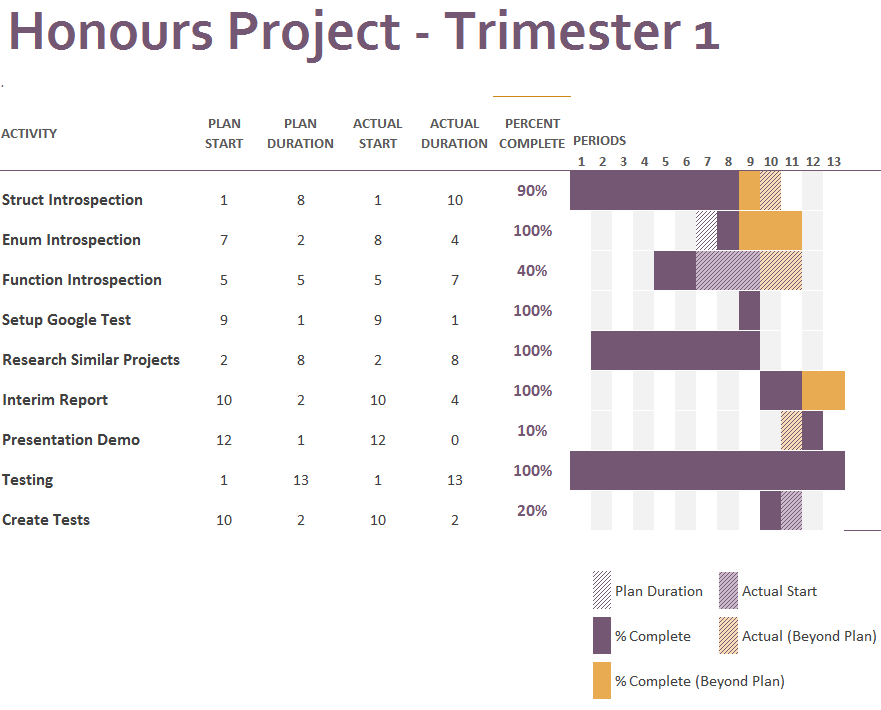
The project was mostly on the Windows operating system, although a lot of it was written on Linux, Ubuntu Distribution. The bulk of the project was written in the *Sublime Text 3* (Skinner 2016) text editor. This was because it is cross platform and runs the same on both Windows and Linux. *Visual Studio* was used to debug the project on Windows, and *CGDB*, a graphical frontend to the *GDB* debugger, was used to debug on Linux. **Reference all tools.**

## Version Control

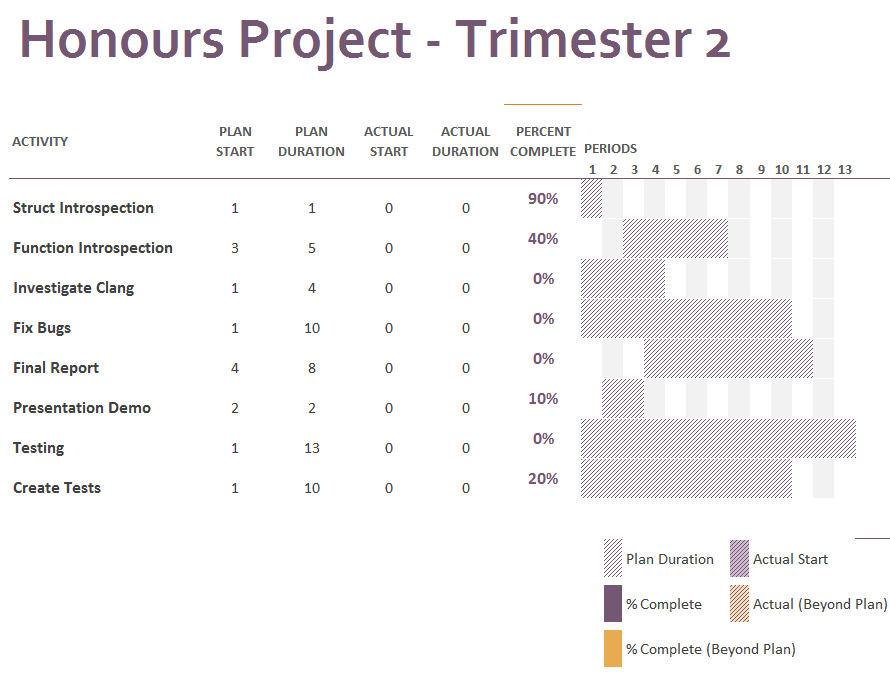
The project used *Git* as its primary version control system, storing the code on the website *GitHub*. The code is available here: <https://github.com/CaptainSeagull/Preprocessor>.

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. At the time of writing, there have been 248 commits, which show that the project has been developed in rapid, small chunks. This allows older commits to be viewed in somewhat isolation, because the commits are generally small and only focus on one thing.

## 2.4 Gantt charts



*Figure ??: Gantt chart of progress for Trimester 1.*



*Figure ??: Gantt chart of progress for Trimester 2.*

**Change Gantt chart 2**

## 2.5 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. **CAN I REFERENCE THE AGILE-NESS??**

# 3.0 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.

## 3.1 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.

|  |
| --- |
| Using System;  Namespace TestApplication {  Class Program {  Static void Main(string[] args) {  int i=0, j=0;  float f=0.0f;  Type i\_type = i.GetType();  Type j\_type = j.GetType();  Type f\_type = f.GetType();  If(i\_type.Equals(j\_type) == true) {  Console.WriteLine(“I and j are the same type.”);  }  If(i.type.Equals(f\_type) == false) {  Console.WriteLine(“I and f are not the same type.”);  }  /\* Prints:  “i and j are the same type.”  “i and f are not the same type.”\*/  }  }  } |

*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.

|  |
| --- |
| Using System;  namespace TestApplication {  class TestClass {  public int I { get; set; }  public string str { get; set; }  }  class Program {  static void Main() {  TestClass test = new TestClass();  test.i = 10;  test.str = “Hello World”;  foreach(var prop in test.GetType().Getproperties()) {  Console.WriteLine(“{0} : {1}”,  prop.name,  prop.GetValue(test, null));  }  /\* Prints:  “i : 10”  “str : Hello World”\*/  }  }  } |

*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

**Could I change this to compile-time reflection in other languages?**

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 function, which is built into the language, 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.

|  |
| --- |
| import std.stdio;  struct A { int a; }  void main() {  A a;  writeln(\_\_traits(hasMember, A, “a”)); // true  writeln(\_\_traits(hasMember, A, “b”)); // false    // true (sizeof declared implicitly)  writeln(\_\_traits(hasMember, A, “sizeof”));  } |
| import std.stdio;  class Test {  private:  int a;  public:  void set\_a(int i) { a = i; }  int get\_a() { return a; }  }  void main() {  auto all\_members = [\_\_traits(allMembers, Test)];  writeln(all\_members);  /\* [“a”, “set\_a”, “get\_a”, “toString”, “toHash”,  “opCmp”, “opEquals”, “Monitor”, “factory”] \*/  } |
| import std.stdio;  struct Test { int a; }  void main() {  // Test whether some code will actually compile or not.  // Useful for templates.  writeln(\_\_traits(compiles, Test));  writeln(\_\_traits(compiles, Test + 1));  } |

*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.

|  |
| --- |
| import std.stdio;  void func() {}  void main() {  int var;  writeln(is(typeof(var) == function)); // false  writeln(is(typeof(func) == function)); // true  } |

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

## External Introspection Tools

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 Compiler*, which will from now on be referred to as *Moc* (Oliver, 2016)*.* *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. 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

**Talk about introspection in C++11, like type\_traits and decltype, and in C++17, like Structured Bindings.**

## Future 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.

|  |
| --- |
| #include <reflexpr>  #include <iostream>  struct A {  public:  int a;  private:  int b;  }  int main(int argc, char \*\*argv) {  typedef reflexpr(A) meta\_A;  std::cout << “The number of public data members is “ <<  std::meta::get\_size\_v<std::meta::get\_data\_members\_t<meta\_A>>;  std::cout << ‘\n’;  std::cout << “The total number of data members is “ <<  std::meta::get\_size\_v<std::meta::get\_all\_data\_members\_t<meta\_A>>;  /\* Output:  The number of public data members is 1  The total number of data members is 2 \*/  } |

*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 assumes 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 tool 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 code has been tested in C++14 and C++17-complient compilers, and will work fine with them. The tool does not require C++14 or C++17 to run, however, because these versions are still new, and forcing people to have them to run the tool would have limited the number of people who could use the tool.

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*. This was done because, on Linux shared libraries generally work well, on Windows they do not. On windows, most applications must ship with whatever version of the *CRT*, and if that gets updated it may break their code.

The tools 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.

## Usage

The pre-processor is just a small command-line tool which takes some arguments. It is just 444 KB large, and runs significantly faster than any modern C++ compiler. **Add some times compared to GCC, Clang and MSVC.**

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

|  |
| --- |
| preprocessor test\_code.cpp  g++ test\_code.cpp |

*Figure ??: Example using the preprocessor tool with GCC.*

The first line, *preprocessor* *test\_code.cpp*, calls the tool on a sample program. This will generate a directory, *pp\_generated*, and two files, *static\_generated.h*, and, *test\_code\_generated.h*. The first file, *static\_generated.h*, does not change between runs and is always written out the exact same. 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 of figure ??, *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.

|  |
| --- |
| preprocessor test\_code\_one.cpp test\_code\_two.cpp  g++ test\_code\_one.cpp test\_code\_two.cpp |

*Figure ??: Example using the preprocessor tool, passing in two files.*

Figure ?? shows an example of using the tool will two files, *test\_code\_one.cpp* and *test\_code\_two.cpp*. Figure ?? will still generate the directory *pp\_generated* and *static\_generated.h*, like figure ??, but it will generate two meta files now, *test\_code\_one\_generated.h*, and *test\_code\_two\_generated.h*. These two generated files should be included in the relevant files.

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

* Gain introspection data on classes at compile time, such as; their base class, the number of members the class has, and a way to convert the class type into a string.
* Ways to get the members of a class by an index, and ways to iterate through a class to access every member.
* Get the number of elements in an enum, at compile time.
* Convert a string into an enum index at runtime.
* Convert an enum index into a string at runtime.

## 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, for *Errors*, then the tool will output errors to the console.

If the user passes the flag –h in, for *Help*, 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, and are compiled out for release builds. They are noted here for completeness.

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, for Tests*, then the program will run all the 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.

|  |
| --- |
| preprocessor test\_code.cpp –e -t |

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

## Google Test

The Google Test framework was used in order to test the parser, and find bugs quickly. Using it allowed me to perform large changes to the codebase, 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.

|  |
| --- |
| TEST(StructText, number\_of\_members\_test) {  char \*str = “struct A { int a, b, c; };”  StructData gen = parse\_struct\_test(str);  ASSERT\_TRUE(gen.member\_count == 3)  << “Error: Number of members in struct is not correct.”;  } |

*Figure ??: 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.

The release build of the application does not link to Google Test, in order to keep the executable size down.

## 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.

**Write about how I tried to use Clang to parse the code, but it was too shitty to use.**

## Note on names in sections, ??, ??, ??. **(The ones below!**

The C++ specification is sometimes a little loose on terms. An example would be, from figure ??, what is an *int* to an *int \*?*

|  |
| --- |
| int integer;  int \*ptr;  int &ref;  int arr[32]; |

*Figure ??: A simple pointer example in C++.*

If we dereference *ptr*, then it’s type becomes an *int*. However, the C++ standard does not have a well-defined term for what an *int* is to an *int \**, an *int &*, or an *int* array. In the tool, I have defined an *int* as a *weak type* of an *int \**, as it is the same type, but without any specifier.

## TypeInfo specialization

**Re-write parts of this with the new TypeInfo struct.**

The generated code has a special templated class called TypeInfo. The implementation is shown in figure ??.

|  |
| --- |
| template<typename T> class pp::TypeInfo {  public:  using type = void;  using weak\_type = void;  using base = void;  static constexpr const char \* const name = NULL;  static constexpr const char \* const weak\_name = NULL;  static constexpr const size\_t member\_count = 0;  static constexpr const bool is\_ptr = false;  static constexpr const bool is\_ref = false;  static constexpr const size\_t base\_count = 0;  static constexpr const bool is\_primitive = false;  static constexpr const bool is\_class = false;  static constexpr const bool is\_enum = false;  }; |

*Figure ??: TypeInfo implementation, located within static\_generated.h.*

This implementation is the default that is used for whenever the user wants to get introspection information about a type. The generated code will scan all the classes within the file and will specialize generations of this class for each implementation.

A subtle design decision, worth noting, is that I set the *type* field to *void* in the default specification, not *T*. While setting it to *T* may help make the code more robust, if, for instance, the user wanted to introspect a class in a head file the system had missed, then this could work. However, most of the other fields would give misleading information. Because of this, a conscious design decision was made to set everything to obviously-wrong values, so that the user would notice the bug and could report it, rather than the system attempting to mask the bug when it should simply be fixed.

**Clang complains when the constexpr strings aren’t const. Is this due to something in the language, or is Clang being stupid??**

Figure ?? shows an example of a typical class, and how the TypeInfo specialization would be generated for it.

|  |
| --- |
| class BaseClass {  public:  float x;  float y;  float z;  };  class SomeClass : public BaseClass {  public:  int a;  int b;  int c;  }; |

*Figure ??: Simple struct example*

|  |
| --- |
| template<> class pp::TypeInfo<SomeClass> {  public:  using type = SomeClass;  using weak\_type = SomeClass;  using base = BaseClass;  static constexpr const char \* const name = "SomeClass";  static constexpr const char \* const weak\_name = "SomeClass";  static constexpr size\_t member\_count = 3;  static constexpr const bool is\_ptr = false;  static constexpr const bool is\_ref = false;  static constexpr const size\_t base\_count = 1;  static constexpr const bool is\_primitive = false;  static constexpr const bool is\_class = true;  static constexpr const bool is\_enum = false;  }; |

*Figure ??: Template specialization of TypeInfo for SomeClass*

Having this template specialization, using static members, means that the user can quickly query information about a class and the information is generated at compile time.

In the actual tool, six specializations are generated for each class. The first line of each specialization is shown in figure ??, but, for the sake of brevity, only the first line is shown. Each of the relevant fields, such as *is\_ptr* and *is\_ref*, and obviously changed to match the version.

|  |
| --- |
| Template<> class pp::TypeInfo<SomeClass>  template<> class pp::TypeInfo<SomeClass \*>  template<> class pp::TypeInfo<SomeClass \*\*>  template<> class pp::TypeInfo<SomeClass&>  template<> class pp::TypeInfo<SomeClass \*&>  template<> class pp::TypeInfo<SomeClass \*\*&> |

*Figure ??: The six generated specializations of pp::TypeInfo for SomeClass.*

The first field, the *type* field, is just a *typedef*, using the C++11 *using* *typedef* keyword **(Reference!)**, to the type passed in. The second field, *weak\_type*, is the type of the class without any qualifiers. If the user passed in a pointer, for example, then this will just be the vanilla type of the class without the pointer. While this may seem redundant, especially since there are two fields the same for the non-pointer version, the code is designed to be flexible. Complicated C++ libraries often make heavy use of templates, and being able to query not just what the type passed in is, but what its base is, is useful.

Figure ?? shows an example of using TypeInfo::weak\_type. The nice thing about this example is how robust it is. If *i* is changed from an *int \** to an *int \*\** or just an *int*, then the type of *k* will remain the same.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  int main(int argc, char \*\*argv) {  int \*i;  pp::TypeInfo<decltype(i)>::type j;  pp::TypeInfo<decltype(i)>::weak\_type k;  return(0);  } |

*Figure ??: An example of using TypeInfo::weak\_type. In the example, i and j are int \*’s, and k is an int.*

The next two fields declared are *name* and *weak\_name*. Like *type* and *weak\_type*, these correspondent to the actual type and the base version of the type, except as strings. These are useful for outputting debug information about a type, or could also be used for writing a type to disk. A simple example using these is shown in figure ??.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  int main(int argc, char \*\*argv) {  int \*i, j;  std::cout << pp::TypeInfo<decltype(i)>::name << std::endl; // "int \*"  std::cout << pp::TypeInfo<decltype(j)>::name << std::endl; // "int"  std::cout << pp::TypeInfo<decltype(i)>::weak\_name << std::endl; // "int"  std::cout << pp::TypeInfo<decltype(j)>::weak\_name << std::endl; // "int"  return(0);  } |

*Figure ??: Example using the name and weak\_name fields of TypeInfo. Because i is an int \*, and j is an int, the names are different for them; but the weak\_name is the same.*

The field *is\_ptr* on the TypeInfo struct is just a boolean to tell if something is a pointer. Figure ?? shows a basic example of when this could be useful in a templated function.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  template<typename T>void do\_something(T var) {  if(pp::TypeInfo<T>::is\_ptr) {  std::cout << "This is a pointer" << std::endl;  } else {  std::cout << "This is not a pointer" << std::endl;  }  }  int main(int argc, char \*\*argv) {  int \*i, j;  do\_something(i); // "This is a pointer"  do\_something(j); // "This is not a pointer"  return(0);  } |

*Figure ??: An example of when the is\_ptr field of TypeInfo could be useful.*

Another example of when it would be useful to test if something is a pointer would be in templated functions that can take either one. Because, in C++, members of a class pointer must be dereferenced and then accessed, done with the *->* operator, while normal classes can only be accessed using the dot (*.*) operator, this can prove problematic if the user wishes to accept either. Figure ?? shows an example that helps solve this problem, using *C++17* *constexpr if* statements. Because *is\_ptr* is a compile-time constant, it can be used with the *constexpr if* statements to do things like this. In fact, every member of the *TypeInfo* struct is a constant, except the strings *name* and *weak\_name*, so you could do a *constexpr* *if* switch on any of the members and it would compile.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  template<typename T>void do\_something(T var) {  if constexpr(pp::TypeInfo<T>::is\_ptr) {  std::cout << "Var is a pointer. ";  std::cout << "Text is " << var->text << std::endl;  } else {  std::cout << "Var is not a pointer. ";  std::cout << "Text is " << var.text << std::endl;  }  }  class String {  public:  char \*text;  int length;  };  int main(int argc, char \*\*argv) {  String str, \*str\_ptr;  str.text = "hello";  str.length = strlen(str.text);  str\_ptr = new String;  str\_ptr->text = "world";  str\_ptr->length = strlen(str\_ptr->text);  do\_something(str); // "Var is a pointer. Text is hello"  do\_something(str\_ptr); // "Var is not a pointer. Text is world"  return(0);  } |

*Figure ??: An example of using the is\_ptr field of TypeInfo and doing a constexpr if on it. While there is still some code duplication, do\_something will work whether a pointer is passed to it or not.* ***TODO(Jonny): This is untested, because MSVC don’t have a C++17 compiler yet… so test it on Clang later.***

The *base\_count* fields of *TypeInfo* is just an integer which tells the user how many classes the class templated on inherits from.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class BaseOne {};  class BaseTwo {};  class BaseThree {};  class Test : public BaseOne, public BaseTwo, public BaseThree {};  int main(int argc, char \*\*argv) {  Test test;  std::cout << pp::TypeInfo<Test>::name << " inherits from " <<  pp::TypeInfo<Test>::base\_count << " classes.";  // “Test inherits from 3 classes.”  return(0);  } |

*Figure ?? Shows an example of how to use the base\_count field of TypeInfo to get the number of classes inherited from.*

The example in figure ?? could be taken further, in order to develop a generic function which can print how many classes any class passed into it inherits from.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class BaseOne {};  class BaseTwo {};  class BaseThree {};  class Test : public BaseOne, public BaseTwo, public BaseThree {};  template<typename T>void print\_base\_class\_count(T var) {  std::cout << pp::TypeInfo<T>::name << " inherits from " <<  pp::TypeInfo<T>::base\_count << " classes" <<  std::endl;  }  int main(int argc, char \*\*argv) {  Test test;  BaseOne test2;  print\_base\_class\_count(test); // “Test inherits from 3 classes.”  print\_base\_class\_count(test2); // “BaseOne inherits from 0 classes.”  return(0);  } |

*Figure ??: Shows a more generic version of using the base\_count field of TypeInfo. The generic function will print out how many classes any class passed into it inherits from.*

The final field of the *TypeInfo* struct is a typedef of the inherited class, called *base*. If the class does not inherit from anything, this is set to *void*. Otherwise, this is set to the name of the first class inherited from. While C++ supports multiple inheritance, it is not as commonly used as single inheritance. And due to some limitations on how the *using* keyword can be used, it was decided that only the first inherited-from class will be available.

Figure ?? shows an example of using the *base* field in order to create an instance of the base class.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class Base {};  class Test : public Base {};  int main(int argc, char \*\*argv) {  Test test;  pp::TypeInfo<decltype(test)>::base test\_base;  // “Test”  std::cout << pp::TypeInfo<decltype(test)>::name << std::endl;    // “Base”  std::cout << pp::TypeInfo<decltype(test\_base)>::name << std::endl;  return(0);  } |

*Figure ??: An example of creating a base class using the base field of the TypeInfo struct. The type of the original class, and the base class, are then printed to the console.*

The example in figure ?? can be done without even creating an instance of the base class. This is shown below in figure ??.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class Base {};  class Test : public Base {};  int main(int argc, char \*\*argv) {  Test test;  // "Test"  std::cout << pp::TypeInfo<decltype(test)>::name  << std::endl;    // “Base”  std::cout << pp::TypeInfo<pp::TypeInfo<decltype(test)>::base>::name  << std::endl;  return(0);  } |

*Figure ??: An example of printing the base type of a class without ever having made an instance of that type.*

Using the *base* field of *pp::TypeInfo* within itself, you can go up a chain of inherited classes to get the one at the top. Figure ?? shows this.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class BaseTwo {};  class BaseOne : public BaseTwo {};  class Test : public BaseOne {};  int main(int argc, char \*\*argv) {  // The inheritance hierarchy is:  // BaseTwo -> BaseOne -> Test  char const \*str = pp::TypeInfo<  pp::TypeInfo<pp::TypeInfo<Test>::base>::base  >::name;    // Prints "BaseTwo"  std::cout << str;  return(0);  } |

*Figure ??: Goes up the inheritance hierarchy by putting templates-within-templates, in order to print out the class that Test’s base-class inherits from.*

Making the example in figure ?? even more generic, using C++17’s *constexpr if*, which is required so that an infinite-recursion of templates does not happen, we can write a generic function which will find the base type of any class and print it. Figure ??, which requires a C++17-compliant compiler to run, shows the generic example of this. The recursion stops when *TypeInfo::base*’s type is *void*, which is the default it is set to and is a primitive type that cannot be inherited from.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class BaseTwo {};  class BaseOne : public BaseTwo {};  class Test : public BaseOne {};  template<typename T>void chain() {  if constexpr(pp::type\_compare(pp::TypeInfo<T>::base, void)) {  std::cout << "The most base classes type is " <<  pp::TypeInfo<T>::name << std::endl;  } else {  chain<pp::TypeInfo<T>::base>();  }  }  int main(int argc, char \*\*argv) {  chain<Test>(); // Prints “BaseTwo”.  chain<BaseOne>(); // Prints “BaseTwo”.  return(0);  } |

*Figure ??: Generate recursive templates in order to print what the most base class is in an inheritance hierarchy.*

The code can also work hand-in-hand with the *type\_traits* library. Figure ?? shows an example of getting the base class using *pp::TypeInfo*, but statically asserting it using type traits *std::is\_base\_of*.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <type\_traits>  class Base {};  class Test : public Base {};  int main(int argc, char \*\*argv) {  static\_assert(std::is\_base\_of<pp::TypeInfo<Test>::base, Base>::value,  "This assert will not get thrown, because Base is the "  "base class of Test");  return(0);  } |

*Figure ??: An example of getting a base class using pp::TypeInfo::base, but asserting that it was correct as compile type by using std::is\_base\_of.*

## Get Member Information

The system also allows the user to get information on members of a class based on its index. The function definition for this is shown in figure ??.

|  |
| --- |
| MEMBER\_TYPE \* pp::get\_member(CLASS\_TYPE \*variable, size\_t index); |

*Figure ??: pp::get\_member definition.*

Due to some limitations on how types can work in C++, the code to access members by index is a little verbose. In C++, you cannot template the return type of a function from within the function, because the calling code must know the size before calling the functions **(Reference)**. Because of this, it can make getting the return type of a function difficult. The following code, which will not compile, shows how this could be done in some languages where type information is provided at runtime.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  class Test {  public:  int a;  float f;  short s;  bool b;  };  int main(int argc, char \*\*argv) {  Test test = {10, 3.14f, 4, true};  // Iterate through each member.  for(int i = 0; (i < pp::TypeInfo<decltype(test)>::member\_count); ++i) {  // Reference to the member at index i.  auto member = pp::get\_member(&test, i);  // Print out the member’s type and the value its holding.  std::cout << pp::TypeInfo<member>::name << " " <<  member << std::endl;  }  return(0);  } |

*Figure ??: An example of how type information might be used if C++ was a higher level language which had type information at runtime. However, due to some limitations in the language, this code will not compile.*

To solve the problem of not being able to *truly* iterate through members, templates can be used in order to generate the relevant serialization code. The rest of this section will continue to discuss and set this up.

Figure ?? starts off with some basic examples of *pp::get\_member*. The function *pp::get\_member* is specialization for each class, and then again for each member of a class. Because of this, the return type will be different, depending on the index passed in.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class Test {  public:  int i; float f; double d; bool b;  };  int main(int argc, char \*\*argv) {  Test test;  test.i = 10;  test.f = 3.14f;  test.d = 3.1415;  test.b = true;  auto i = pp::get\_member(&test, 0); // i is an int \*.  auto f = pp::get\_member(&test, 1); // f is a float \*.  auto d = pp::get\_member(&test, 2); // d is an double \*.  auto b = pp::get\_member(&test, 3); // b is an bool \*.  // All of these assets will be true.  assert(i == &test.i);  assert(f == &test.f);  assert(d == &test.d);  assert(b == &test.b);  return(0);  } |

*Figure ??: Basic get\_member example.*

The function *pp::get\_member* will also works for pointers. In this case, the return type will be a pointer to that pointer. For instance, if a class had a member that was an *int \**, then *pp::get\_member* would return an *int \*\** for that index. Figure ?? shows an example of this, as well an example of a class within a class.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class V2 {  public:  int x, y;  };  class Test {  public:  int \*i; V2 v;  };  int main(int argc, char \*\*argv) {  Test test;  // Example where the member is a pointer.  test.i = new int;  \*test.i = 10;  int \*\*i = pp::get\_member(&test, 0);  std::cout << \*test.i << ‘ ‘ << \*\*i; prints “10 10”.  // Example where the member is another class.  test.v = {2, 4};  auto v = pp::get\_member(&test, 1);  auto x = pp::get\_member(v, 0);  auto y = pp::get\_member(v, 1);  // Both of these will be true  std::cout << test.v.x << ‘ ‘ << \*x; // Prints “2 2”.  std::cout << test.v.y << ‘ ‘ << \*y; // Prints “4 4”.  return(0);  } |

*Figure ??: An example that shows introspecting a class within a class, and a pointer within a class.*

While figure ?? does show an example of iterating through a known class, it is much more valuable to be able to iterate through an unknown class. Figure ?? shows how, using the preprocessor, you can write generic serialization code in a fairly generic way. In the example, we write the contents of two classes, *TestOne*, and *TestTwo*, to the console.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class TestOne {  public:  int i; float f;  };  class TestTwo {  public:  double d; bool b;  };  template<typename T, int index>void print\_var(T \*var) {  print\_var<T, index - 1>(var);  auto member = pp::get\_member(var, index);  /\* This is weak name, because pp::get\_member returns a pointer  to the member. \*/  char const \*type\_as\_str = pp::TypeInfo<decltype(member)>::weak\_name;  std::cout << type\_as\_str << ' ' << \*member << std::endl;  }  /\* Empty specializations, so that print\_var doesn’t  recursively generate infinite functions.\*/  template<>void print\_var<TestOne, -1>(TestOne \*t) {}  template<>void print\_var<TestTwo, -1>(TestTwo \*t) {}  // Simple utility function to make calling print\_var nicer.  template<typename T> void my\_print(T \*var) {  print\_var<T, pp::TypeInfo<T>::member\_count - 1>(var);  }  int main(int argc, char \*\*argv) {  //  // TestOne.  //  TestOne test\_one;  test\_one.i = 10;  test\_one.f = 3.14f;  /\* Prints "int 10  float 3.14" \*/  my\_print(&test\_one);  //  // TestTwo.  //  TestTwo test\_two;  test\_two.d = 3.1415;  test\_two.b = true;  /\* Prints "double 3.1415  bool 1" \*/  my\_print(&test\_two);  return(0);  } |

*Figure ??: Basic generic serialization example.*

In the example, we first call *my\_print*. Inside *my\_print*, it calls the function *print\_var*, and sets up all the type information *my\_var* needs. Because some of this introspection code is boilerplate, and is easy to get wrong, it is nicer to wrap it up in a hard-to-mess-up interface.

The first line of *print\_var* generates another template call to *print\_var*, for one minus the member count. This recursively goes down until the index passed in is *-1*, at which point the specialization of *print\_var* for *-1* is called, which does nothing. This is necessary so *print\_var* doesn’t recursively call itself forever.

Inside *print\_var*, we use *pp::get\_member* to get a pointer to the member at an index. Because *pp::get\_member* can have different return types, the variable *member* must be declared as *auto*. We then, using *std::cout*, print the type of the member at an index, and the value it holds.

If you are using a C++17 compatible compiler, then this can be taken further. Using *constexpr if* statements, then you can change this code to work for a class within a class. Figure ??, which will only compile under a C++17 compile, like Clang **(Find out the versions for Clang, MSVC, and GCC)**, shows this.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class V2 {  public:  int x, y;  };  class Test {  public:  int i; V2 v;  };  // Must be forward declared, so print\_var can find it.  template<typename T> void my\_print(T \*var);  template<typename T, int index>void print\_var(T \*var) {  print\_var<T, index - 1>(var);  auto member = pp::get\_member(var, index);  if constexpr(pp::TypeInfo<decltype(member)>::is\_primitive) {  print\_var<T, index - 1>(var);  /\* This is weak name, because pp::get\_member returns a pointer  to the member. \*/  char const \*type\_as\_str = pp::TypeInfo<decltype(member)>::weak\_name;  std::cout << type\_as\_str << ' ' << \*member << std::endl;  } else {  my\_print(\*member);  }  }  /\* Empty specializations, so that print\_var doesn’t  recursively generate infinite functions.\*/  template<>void print\_var<Test, -1>(Test \*t) {}  // Simple utility function to make calling print\_var nicer.  template<typename T> void my\_print(T \*var) {  print\_var<T, pp::TypeInfo<T>::member\_count - 1>(var);  }  int main(int argc, char \*\*argv) {  Test test;  test.i = 10;  test.v = {2, 4};  /\* Prints "int 10  int 2  int 4" \*/  my\_print(&test);  return(0);  } |

*Figure ??: A more generic version of figure ??, which will work with classes-within-a-class. Will only compile under C++17 though.*

Figure ?? demonstrates how this can be used to work with pointers as well. Like ??, however, this requires *constexpr if* statements, and is only compatible with a C++17 compiler.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class Test {  public:  int i;  int \*ptr;  };  template<typename T, int index>void print\_var(T \*var) {  print\_var<T, index - 1>(var);  auto member = pp::get\_member(var, index);  char const \*type\_as\_str = pp::TypeInfo<decltype(member)>::weak\_name;  /\* If member is a pointer-to-a-pointer, then it means  that the member variable was a pointer. If member  is just a pointer, then tha member variable was a  normal type.\*/  if constexpr(pp::TypeInfo<decltype(\*member)>::is\_ptr) {  std::cout << type\_as\_str << ' ' << \*\*member << std::endl;  } else {  std::cout << type\_as\_str << ' ' << \*member << std::endl;  }  }  /\* Empty specializations, so that print\_var doesn’t  recursively generate infinite functions.\*/  template<>void print\_var<Test, -1>(Test \*t) {}  // Simple utility function to make calling print\_var nicer.  template<typename T> void my\_print(T \*var) {  print\_var<T, pp::TypeInfo<T>::member\_count - 1>(var);  }  int main(int argc, char \*\*argv) {  Test test;  test.i = 10;  test.ptr = new int;  \*test.ptr = 5;  my\_print(&test);  return(0);  } |

*Figure ??: A generic print statement which will work with pointers. Only compatibly with C++17.*

Finally, figure ?? combines both figures ?? and ??, in order to write a generic printing function, which will handle classes within classes, and pointers.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class Test {  public:  int i;  int \*ptr;  };  // Must be forward declared, so print\_var can find it.  template<typename T> void my\_print(T \*var);  template<typename T, int index>void print\_var(T \*var) {  print\_var<T, index - 1>(var);  auto member = pp::get\_member(var, index);  if constexpr(pp::TypeInfo<decltype(member)>::is\_primitive) {  char const \*type\_as\_str = pp::TypeInfo<decltype(member)>::weak\_name;  /\* If member is a pointer-to-a-pointer, then it means  that the member variable was a pointer. If member  is just a pointer, then tha member variable was a  normal type.\*/  if constexpr(pp::TypeInfo<decltype(\*member)>::is\_ptr) {  std::cout << type\_as\_str << ' ' << \*\*member << std::endl;  } else {  std::cout << type\_as\_str << ' ' << \*member << std::endl;  }  } else {  if constexpr(pp::TypeInfo<decltype(\*member)>::is\_ptr) {  my\_print(\*\*member);  } else {  my\_print(\*member);  }  }  }  /\* Empty specializations, so that print\_var doesn’t  recursively generate infinite functions.\*/  template<>void print\_var<Test, -1>(Test \*t) {}  // Simple utility function to make calling print\_var nicer.  template<typename T> void my\_print(T \*var) {  print\_var<T, pp::TypeInfo<T>::member\_count - 1>(var);  }  int main(int argc, char \*\*argv) {  Test test;  test.i = 10;  test.ptr = new int;  \*test.ptr = 5;  my\_print(&test);  return(0);  } |

*Figure ??: A fully generic printing function, which is written in user code. Only compatible with a C++17 compiler, as it uses constexpr if statements.*

The nice thing about this example is all the seriation is in user code, not behind a black box, like *pp::print* function, which is discussed in figure ??.

Building on the previous examples even more, figure ?? write a class to disk. It write a class to disk in XML **(Reference)** format, which could be read by other tools in order to draw some data about the class.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  #include <sstream>  class Test {  public:  int i; float f;  };  template<typename T, int index> void serialize\_var(T \*var,  std::stringstream&buffer) {  serialize\_var<T, index - 1>(var, buffer);  auto member = pp::get\_member(var, index);  // Type of the member.  char const \*type\_as\_str = pp::TypeInfo<decltype(member)>::weak\_name;  // Name of the member.  char const \*member\_name = pp::get\_member\_name<T>(index);  // Output member in xml format.  buffer << " <name>" << member\_name << "</name>" << std::endl;;  buffer << " <type>" << type\_as\_str << "</type>" << std::endl;;  buffer << " <value>" << \*member << "</value>" << std::endl;;  }  // Empty specializations, so that serialize\_var doesn’t  // recursively generate infinite functions.  template<>void serialize\_var<Test, -1>(Test \*t, std::stringstream &buffer) {}  // Simple utility function to make calling serialize\_var nicer.  template<typename T> void write\_to\_xml(T \*var, std::string name) {  // Write to string stream.  std::stringstream buffer;  buffer << "<" << pp::TypeInfo<T>::weak\_name << ">" << std::endl;  serialize\_var<T, pp::TypeInfo<T>::member\_count - 1>(var, buffer);  buffer << "</" << pp::TypeInfo<T>::weak\_name << ">" << std::endl;  // Write to disk.  name = name + ".xml";  FILE \*file = fopen(name.c\_str(), "w");  if(file) {  fwrite(buffer.str().c\_str(), 1, buffer.str().size(), file);  fclose(file);  }  }  int main(int argc, char \*\*argv) {  Test test;  test.i = 10;  test.f = 3.14f;  write\_to\_xml(&test, "test");  return(0);  } |

*Figure ??: An example of serializing a class into XML format.*

|  |
| --- |
| <Test>  <name>i</name>  <type>int</type>  <value>10</value>  <name>f</name>  <type>float</type>  <value>3.14</value>  </Test> |

*Figure ??: The example from figure ??, in XML format.*

## 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.

|  |
| --- |
| if(int == int) |

*Figure ??: 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.

|  |
| --- |
| int i;  if(decltype(i) == int) |

*Figure ??: Invalid type comparison using decltype.*

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

|  |
| --- |
| template<typename T>  void foo(T a) {  if(decltype(a) == int) {  // Do integer stuff.  } else if(decltype(a) == float) {  // Do float stuff.  }  } |

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

The metaprogramming tool, however, exposes three mechanism for comparing types. Figure ?? shows the definition for this function, *pp::type\_compare*, which is used to evaluate whether two types are the same at compile time. This function is semantically the same as *std::is\_same*, and either one could be used.

|  |
| --- |
| bool pp::type\_compare(TYPE a, TYPE b); |

*Figure ??: pp::type\_compare function definition.*

Using *pp::type\_compare* with *pp::TypeInfo* can greatly simplify metaprogramming. Figure ?? shows an example comparing an *int \** and an *int*.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  int main(int argc, char \*\*argv) {  int i, \*j;  if(pp::type\_compare(pp::TypeInfo<decltype(i)>::weak\_type,  pp::TypeInfo<decltype(j)>::weak\_type)) {  std::cout << "i and j have the same base type!" << std::endl;  }  return(0);  } |

*Figure ??: An example of using pp::type\_compare with pp::TypeInfo to compare an int \* and an int.*

## 

## Print Class

One of the most powerful methods available inside the preprocessor is used for printing aclassto the console. The function definition is provided in figure ??.

|  |
| --- |
| void pp::print(TYPE v, char \*buffer = NULL, size\_t buffer\_size = 0); |

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

The function takes three parameters. The first is the variable that you wish to print. The second and third are optional parameters; which are the length and size of a buffer that the user can pass in. If these are left to their default values, then the function will allocate and free the memory for printing the struct itself.

Figure ?? shows a very simple example of serializing a class and printing it to the console.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class String {  public:  char \*text;  int length;  };  int main(int argc, char \*\*argv) {  String str;  str.text = "Hello World";  str.length = strlen(str.text);  pp::print(str);  /\* Prints:  "String str  char \*text = "Hello World"  int length = 11"\*/  return(0);  } |

*Figure ??: A simple example using pp::print.*

The function *pp::print* goes through all the members of the struct and prints them. In figure ??, it first goes to the member *text* and prints that, then it prints the member *length*.

Figure ?? shows a more complex example, and how the *pp::print* method handles it.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  class Vector2 {  public:  int x, y;  };  class Test {  public:  int i;  float f;  Vector2 v2;  int \*i\_ptr;  float \*f\_ptr;  Vector2 \*v2\_ptr;  int i\_arr[2];  float f\_arr[3];  };  int main(int argc, char \*\*argv) {  Test test;  test.i = 1;  test.f = 2.5f;  test.v2 = {3, 4};  test.i\_ptr = NULL; // Intentionally set to null  test.f\_ptr = new float; \*test.f\_ptr = 5.25f;  test.v2\_ptr = new Vector2; \*test.v2\_ptr = {6, 7};  for(int i = 0; (i < 2); ++i) test.i\_arr[i] = i;  for(int i = 0; (i < 3); ++i) test.f\_arr[i] = i;  pp::print(test);  /\* Prints:  "Test test  int i = 1  float f = 2.500000  Vector2 v2  int x = 3  int y = 4  int \*i\_ptr = (null)  float \*f\_ptr = 5.250000  Vector2 \*v2\_ptr  int x = 6  int y = 7  int i\_arr[0] = 0  int i\_arr[1] = 1  int f\_arr[0] = 0.000000  int f\_arr[1] = 1.000000  int f\_arr[2] = 2.000000"\*/  return(0);  } |

*Figure ??: A complex example using pp::print that has pointers, arrays, and classes within classes.*

In figure ??, you can see how the *pp::print* method handles different types. The first two types, *int i* and *float f* are simple printed to the console normally. The third type *Vector2 v2* has its type printed, then the serialize function recursively calls itself and prints out the members of *v2*.

For the first pointer in figure ??, *int \*i\_p*, the function outputs that *i* is *NULL*, because it was intentionally set to null. The second pointer, *f\_ptr*, is set to the value it was allocated to, and the type outputted is shown to be *float \*f\_ptr*. The third pointer, Vector2 *v2\_ptr*, is recursively called like *v2* in order to print it’s members.

The first array in figure ??, *int i\_arr[2]*, is initially within a for loop. Inside the function, it’s index and the value stored at that index are printed. The same steps are taken for the second array*, float f\_arr[3]*.

A similar function to *pp::print* is available, *pp::serialize*. It’s definition is shown in figure ??.

|  |
| --- |
| size\_t pp::serialize(TYPE var, char \*buffer, size\_t buffer\_size); |

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

The function *pp::serialize* will fill out the *buffer* variable with the serialized data, rather than print it to the console like *pp::print*. This can be useful if you wanted to write a serialized class to disk.

An example showing when it could be useful to write a serialized class to disk is shown in figure ??, and the output written to disk is shown in figure ??.

|  |
| --- |
| #include “pp\_generated/test\_code\_generated.h”  #include <windows.h>  class Test {  public:  int \*integer[32];  };  int main(int argc, char \*\*argv) {  Test test = {};  \_\_try { // Windows SEH equivalent of “try”.  for(int i = 0; (i < 32); ++i) {  // Skip 15 for this example, so it should be NULL.  if(i == 15) continue;  test.integer[i] = new int;  \*test.integer[i] = i;  }  for(int i = 0; (i < 32); ++i) {  ++(\*test.integer[i]);  }  } \_\_except(1) { // Windows SEH equivalent of “catch”.  size\_t buffer\_size = 1024;  char \*buffer = new char[buffer\_size];  pp::serialize(test, buffer, buffer\_size);  // Create a new file called “test\_output.txt” and write  // the serialized struct to it.  FILE \*file = fopen(“test\_output.txt”, “w”);  if(file) {  fwrite(buffer, 1, buffer\_size, file);  fclose(file);  }  }  return(0);  } |

*Figure ??: A complicated example showing when it could be useful to write a serialized class to disk, using Windows Structed Exception Handling to catch a NULL-pointer dereference.*

|  |
| --- |
| Test test  int \*integer[0] = 1  int \*integer[1] = 2  int \*integer[2] = 3  int \*integer[3] = 4  int \*integer[4] = 5  int \*integer[5] = 6  int \*integer[6] = 7  int \*integer[7] = 8  int \*integer[8] = 9  int \*integer[9] = 10  int \*integer[10] = 11  int \*integer[11] = 12  int \*integer[12] = 13  int \*integer[13] = 14  int \*integer[14] = 15  int \*integer[15] = (null)  int \*integer[16] = 16  int \*integer[17] = 17  int \*integer[18] = 18  int \*integer[19] = 19  int \*integer[20] = 20  int \*integer[21] = 21  int \*integer[22] = 22  int \*integer[23] = 23  int \*integer[24] = 24  int \*integer[25] = 25  int \*integer[26] = 26  int \*integer[27] = 27  int \*integer[28] = 28  int \*integer[29] = 29  int \*integer[30] = 30  int \*integer[31] = 31 |

*Figure ??: The output, which was written to disk, from figure ??.*

Figure ?? shows a good example of writing class information to disk. In figure ??, there is a class that has an array of pointers to integers. Intentionally, for the purpose of the demo, when the pointers memory is allocated, the pointer at index fifteen in the array is left as NULL. Then each index in the array is incremented. When the second *for* loop reaches that element, it attempts to dereference a NULL pointer. Instead of crashing, however, the Structured Exception Handling kicks in and catches the deference. The variable *test* is then serialized into a buffer, and that buffer is written to disk. Looking at the data written to disk, in figure ??, it is obvious that the bug is because the sixtieth element is NULL.

While not every bug would be as obvious to see as the example in figure ??, it should be obvious that having a lot of data serialized to disk during a crash would be useful. It could help find bugs, and could be used in combination with the dump **(FIND A REFERENCE AND MAKE SURE THE FILES ARE CALLED THIS IN MSVC)** files usually generated when something crashed.

## Introspect type from string

Inside the system, there are some utility functions that can generate introspection data for a class, based on a string with the classes name. While these functions are not as useful for simple cases as the *pp::TypeInfo* method, they can come in handy for some trickery cases, where limitations of C++ make it impossible to do something using a type, which can be trivially done using a string of the type.

Figure ?? shows a simple example of a method which converts a string-of-a-type into data. The function, which has had its definition provided in figure ??, can convert a string of a class name into the size required to store it. It works the same as as the *sizeof* operator, except on a string rather than a type.

|  |
| --- |
| static size\_t pp::get\_size\_from\_str(char const \*str); |

*Figure ??: Defintion of pp::get\_size\_from\_str function.*

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  class V3 {  public:  int x;  int y;  int z;  };  int main(int argc, char \*\*argv) {  size\_t a = sizeof(V3);  size\_t b = pp::get\_size\_from\_str("V3");  assert(a == b); // true  return(0);  } |

*Figure ?? Example using pp::get\_size\_from\_str.*

Another string-to-data function is *pp::get\_number\_of\_members\_str*, which will return the number of members in a struct. It’s definition is provided in figure ??, and a basic example is shown in figure ??.

|  |
| --- |
| static size\_t pp::get\_number\_of\_members\_str(char const \*str); |

*Figure ??: Defintion for pp::get\_number\_of\_members\_str.*

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  class V3 {  public:  int x;  int y;  int z;  };  int main(int argc, char \*\*argv) {  size\_t a = 3;  size\_t b = pp::get\_number\_of\_members\_str("V3");  size\_t c = pp::TypeInfo<V3>::member\_count;  // All of these asserts will be true.  assert(a == b);  assert(b == c);  assert(a == c);  return(0);  } |

*Figure ??: Example using pp::get\_numeber\_of\_members\_str.*

**Add more of these string-to-data functions! Do one with base type, which goes up the inheritance tree!**

The reason these string-to-data function exists is because some of the internals of pp::print require them. They can also become useful when dealing with recursively-outputting data. While the *pp::TypeInfo* method for serializing data is recommended, and since a lot of the generate code will be done at compile-time it is generally faster, it has some limitations. For example, there is no way to recursively generate all control paths for serializing a class. The following example shows where the limitations lie.

**Put in the example here, then reference the C++ specification on why it won’t work.**

**Now put in the example using strings, rather than type data.**

## Enums

Enums, defined under some limitations, which are discussed later, can use *pp::TypeInfo* in order to get some information about themselves. Figure ?? shows an example of using an enum class with *pp::TypeInfo* and some of the data you can get from it.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  enum class Letters : short {  a, b, c  };  int main(int argc, char \*\*argv) {  char const \*str = pp::TypeInfo<Letters>::name;  std::cout << str << std::endl; // Prints "Letters".  size\_t n = pp::TypeInfo<Letters>::member\_count;  std::cout << n << std::endl; // Prints "3".  // For enums, base is reused in order to print  // the stored type.  char const \*underlying\_type =  pp::TypeInfo<pp::TypeInfo<Letters>::base>:name;  // Prints "short".  std::cout << underlying\_type << std::endl;  return(0);  } |

There are also two separate functions which are unique to enums, *pp::string\_to\_enum* and *pp::enum\_to\_string*. Both of these function definitions are shown in figure ??. Both of these functions are then specialized for each enum.

|  |
| --- |
| template<typename T> char const \*pp::enum\_to\_string(T element); |
| template<typename T> T pp::string\_to\_enum(char const \*str); |

*Figure ??: The function definitions for pp::enum\_to\_string and pp::string\_to\_enum.*

Using some of the C++ *constexpr* feature from C++17, which relaxes some of the rules for a *constexpr* function, we could calculate both of these at compile time. However, it was chosen to do them at run time because it allows more flexibility, at the cost of some performance. Figure ?? shows some example of *pp::string\_to\_enum*.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <string>  enum Numbers : int {  zero,  one,  two,  };  int main(int argc, char \*\*argv) {  // Using string literal.  Numbers get\_zero = pp::string\_to\_enum<Numbers>("zero");  assert(get\_zero == 0);  // Using std::string.  std::string one\_as\_string = "one";  Numbers get\_one = pp::string\_to\_enum<Numbers>(one\_as\_string.c\_str());  assert(get\_one == 1);  // Using string concatonation.  std::string two\_part\_a = "t";  std::string two\_part\_b = "wo";  std::string full\_two = two\_part\_a + two\_part\_b;  Numbers get\_two = pp::string\_to\_enum<Numbers>(full\_two.c\_str());  assert(get\_two == 2);  return(0);  } |

*Figure ??: some examples using pp::string to enum.*

In the examples in figure ??, it would be impossible to calculate the third example if *pp::string\_to\_enum* was required to work at compile-time, because the data is not there until run time. **C++ constexpr can be run time *or* compile time, so maybe this isn’t true? Paul will know…**

Figure ?? shows some examples of converting an enum into a string. Some of these examples could have been performance at compile time using *constexpr*, but not all of them. Hence, the decision not to use it.

|  |
| --- |
| #include "pp\_generated/test\_code\_generated.h"  #include <iostream>  enum class Numbers : int {  zero,  one,  two,  three  };  int main(int argc, char \*\*argv) {  char const \*zero\_str = pp::enum\_to\_string<Numbers>(Numbers::zero);  std::cout << zero\_str << std::endl; // Prints "zero"  Numbers one\_cpy = Numbers::one;  char const \*one\_str = pp::enum\_to\_string<Numbers>(one\_cpy);  std::cout << one\_str << std::endl; // Prints "One"  int as\_integer = 1;  ++as\_integer;  char const \*two\_str = pp::enum\_to\_string<Numbers>((Numbers)as\_integer);  std::cout << two\_str << std::endl; // Prints "Two"  return(0);  } |

*Figure ??: Examples of converting an enum into a string. In this example, Numbers is an enum class, just to demonstrate that it will work with both a vanilla enum and an enum class.*

The enum introspection data will work with both normal enums and C++11 enum classes. However, there is one important limitation; it will not work with enums that have not had their storage type explicitly defined. Figure ?? demonstrates the difference.

|  |
| --- |
| enum A : int {}; // Supported  enum class B : int {}; // Supported  enum C {}; // Not supported. |

*Figure ??: Different types on enums, and whether their supported or not.*

The reason that the classic c-style enums are not supported is a limitation within the C++ language. In the C++ specification, it states that an enum cannot be forward declared unless it has had its storage type is explicit. This is somewhat constrasting with the C specification, which does allow enums to be forward declares. **Reference both the C++ spec and C spec.**

The code generating from the metaprogramming tool must have all the types forward declared in order to operate on them. Since it cannot forward declare a vanilla enum, it cannot support them. Some compilers do support forward declaring enums without an explicit type, but not all. Because the generated code strictly follows the C++ specification, it does not forward declare them and hence they are not supported.

## Performance

Performance is a big concern for most C++ programmers. C++ is one of the few languages that can compile directory to assembly, with no intermediate byte code layer **(Reference)**. It is also considered an unsafe language, because it can directory access, manipulate, and write over the bounds of memory pages allocated to it **(Reference)**.

Because of these requirements, performance was a large concern when designing this meta programming system. Two examples are given, in figure ?? and ??, respectively, which demonstrate the high performance of the meta programming system, which relies on compile-time constants whenever it can. On the left hand side of each figure is the C++ code, and on the right hand side is the x64 Assembly **(Reference)**, generated by Microsoft Visual Studio 2015 **(Reference)**.

|  |  |
| --- | --- |
| class SomeClass {  public:  int a;  int b;  int c;  int d;  };  int main(int argc, char \*\*argv) {  printf("The number of members in %s are %d", "SomeClass", 4);  return(0);  } | ; main  push ebp  mov ebp,esp  ; printf part  push 4 ; member count  push 1B1F94h ; class name  push 1B1FA8h ; format string  call 00123C3D ; call printf  add esp,0Ch    ; return 0  xor eax,eax  } |

*Figure ??: Simple program which prints a string and an int using printf, and the assembly generated.*

|  |  |
| --- | --- |
| class SomeClass {  public:  int a;  int b;  int c;  int d;  };  int main(int argc, char \*\*argv) {  printf("The number of members in”  “%s are %d",  pp::TypeInfo<SomeClass>::name, pp::TypeInfo<SomeClass>::member\_count);  return(0);  } | ; main  push ebp  mov ebp,esp  ; printf part  push 4 ; member count  push 171F94h ; class name  push 171FA8h ; format string  call 0E3C3Dh ; call printf  add esp,0Ch  ; return 0  xor eax,eax |

*Figure ??: A copy of figure ??, except using the preprocessor to get introspection information.*

As you can see, from figures ?? and ??, the assembly generated is identical in terms of functionality. A nice benefit of having all the members of *pp::TypeInfo* marked as *constexpr* means that, as well as being able to use the information at compile time with complicated algorithms, is that the code assembly generated for them is almost identical to the non-introspection version in terms of performance.

# Future Work.

## Further C++ support.

While the preprocessor tool currently supports a decent subsection of the C++ programming language, it is not complete. While the parser will generally skip over unknown sections of code, it is possible for it to get tripped up and generate incorrect code. A lot of these limitations are due to the complexity of parsing C++ as a language. Some of them are related to complex features, however, like templates or macros.

## Function Introspection.

Right now, there is no function introspection data generated. The parser does currently handle functions, and stores some 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 use-cases for function introspection are much weaker than forclass or enum introspection, so it was not a high priority.

## Error Handling.

Right now, a syntax error in normal code may generate a syntax error in the generated code. And, because the generated 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.

**Re-write, because I do support some C++ containers.**

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.

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 simply 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 significant time to the code generation process.

**Write about how the tool can go hand-in-hand with C++ as it gains more introspection support. Rather than competing with Metaprogramming build-in to the language, it is complimentary and, assuming C++ ever adding good introspection features, could slowly go away a piece at a time.**

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.

# 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>

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>

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

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)

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

GitHub, Inc (2016). *Github website*. Available at: <https://github.com/>

GNU Project (2016). *GNU Compiler Collection*. Available at: <https://gcc.gnu.org/>

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

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

Oliver G. (2016). *Qt’s Moc*.

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

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>

Skinner J. (2016). *Sublime Text 3.* Available at: <http://www.sublimetext.com/>

Unreal Games. (2016). *Unreal Engine Property System.* Documentation available at <https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Reference/Properties/>

**FORM A**

**COMPUTING HONOURS PROJECT (COMP10034)**

**PROGRESS AND MANAGEMENT MEETING AGENDA**

*(To be completed* ***before*** *the scheduled meeting)*

**Student:** Jonathan Livingstone **Supervisor:** Paul Keir

**Meeting Number:** 1 **Date/Time:** 04/10/2016 - 2:00pm

**PROGRESS**

Over the last month, the following tasks have been completed:

* Complete honours spec
* Very basic prototype
* Read up on related literature

**AGENDA FOR FORMAL MEETING (Example)**

* Discuss the progress so far.
* Discuss the Dissertation and Interim Report.
* Setting of tasks and planned targets before next formal meeting.
* Set a date for next formal meeting.

**FORM B**

**COMPUTING HONOURS PROJECT (COMP10034)**

**MANAGEMENT MEETING MINUTES AND PLAN**

*(To be completed* ***after*** *the scheduled meeting)*

**Student:** Jonathan Livingstone **Supervisor:** Paul Keir

**Meeting Number:** 1 **Date/Time:** 04/10/2016 - 2:00pm

**MINUTES**

The following tasks and issues were discussed and specific actions agreed:

* Document the project more, and add a "readme" on the repo page.
* Change the structure of the generated code, so there's a file for "static" code, and each .cpp file in a project gets its own generated file.
* Continue to develop the application so it works with more C++ features.

**PLAN**

The following tasks and timelines have been agreed both for the next month and beyond:

For the next month:

* Have a readme showing how to use the project.
* Have some examples of how the project can be used.
* Have a "static" file, which holds the code that isn't changed.

Beyond the next month

* Work on the application so it can work with a wider range of C++ features.
* Make the file structure so every file parsed gets its own generated file.

**FORM A**

**COMPUTING HONOURS PROJECT (COMP10034)**

**PROGRESS AND MANAGEMENT MEETING AGENDA**

*(To be completed* ***before*** *the scheduled meeting)*

**Student:** Jonathan Livingstone **Supervisor:** Paul Keir

**Meeting Number:** 2 **Date/Time:** 14/11/2016 - 2:00pm

**PROGRESS**

Over the last month, the following tasks have been completed:

* Complete a readme showing how to use the project.
* Have a basic prototype showing an example of how the preprocessor could be used.
* Pulled out non-changing code into a "static" file which isn't generated.

**AGENDA FOR FORMAL MEETING (Example)**

* Discuss the progress so far.
* Discuss next steps for implementation.
* Setting of tasks and planned targets before next formal meeting
* Set a date for next formal meeting.

**FORM B**

**COMPUTING HONOURS PROJECT (COMP10034)**

**MANAGEMENT MEETING MINUTES AND PLAN**

*(To be completed* ***after*** *the scheduled meeting)*

**Student:** Jonathan Livingstone **Supervisor:** Paul Keir

**Meeting Number:** 2 **Date/Time:** 14/11/2016 - 2:00pm

**MINUTES**

The following tasks and issues were discussed and specific actions agreed:

* Discuss the technical progress so far, in comparison to last meeting.
* Discuss the interim report and how to tackle it.

**PLAN**

The following tasks and timelines have been agreed both for the next month and beyond:

For the next month:

* Have a more robust better testing framework.
* Start having some more *tricky* examples of how the framework can be used.
* Complete a draft interim report.
* Create a Gantt chart.
* Track tests and examples used by a competing reflection system.

Beyond the next month

* Continue to develop the application.
* Finish the interim report.

**FORM A**

**COMPUTING HONOURS PROJECT (COMP10034)**

**PROGRESS AND MANAGEMENT MEETING AGENDA**

*(To be completed* ***before*** *the scheduled meeting)*

**Student:** Jonathan Livingstone **Supervisor:** Paul Keir

**Meeting Number:** 3 **Date/Time:** 01/12/2016 - 2:00pm

**PROGRESS**

Over the last month, the following tasks have been completed:

* Added Google Test to the project.
* Added introspection to enums.
* Now generate one .h per project file with static members, for scalability.
* Create a Gantt chart.

**AGENDA FOR FORMAL MEETING (Example)**

* Discussion of literature review.
* Discussion of interim report.
* Discuss further features.
* Set a date for next formal meeting.

**FORM B**

**COMPUTING HONOURS PROJECT (COMP10034)**

**MANAGEMENT MEETING MINUTES AND PLAN**

*(To be completed* ***after*** *the scheduled meeting)*

**Student:** Jonathan Livingstone **Supervisor:** Paul Keir

**Meeting Number:** 3 **Date/Time:** 01/12/2016 - 2:00pm

**MINUTES**

The following tasks and issues were discussed and specific actions agreed:

* How to deal with the interim report.
* Tidy up some of the features a little.

**PLAN**

The following tasks and timelines have been agreed both for the next month and beyond:

For the next month:

* Finish the first draft of the interim report.
* Handle any bugs that come up.

Beyond the next month

* Support for C++ features for introspection.
* Finish the full report.
* Prepare a demo.