UNIVERSITY OF THE WEST of SCOTLAND

UWS

BSc (Hons) Computer Games Technology

Introspection in C++

Jonathan Livingstone
B00236297

24th March 2017

Supervisor: Paul Keir

1 Introduction

1.1 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 allows this to a limited extent, with the use of *type_traits* and the *decltype* and *auto* keywords. C++17 goes even further with the use of *structure* bindings, which can be used to get access to the members of a class, assuming some limitations about it. Despite this, however, they do not allow much more than toy examples and very basic 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 trade-offs 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++ projects with minimal work to set it up on the part of the user, 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.

1.2 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 (Gosling, 1995) and C# (Microsoft, 2000), 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.

1.3 The Problem

Because C++ lacks anything beyond the most basic type 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.

1.4 The Project

This project aims to allow C++ programmers to view their data in a similar ways to other performance-orientated languages, like D (Bright, 2001) or Rust (Hoare, 2010). 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, 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, henceforth referred to as *GCC*, or the Low-Level Virtual Machine, henceforth referred to as *LLVM*. 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.

All of the generated code conforms strictly to the C++11 standard, and should work on Windows, OS X, or Linux operating systems. It has been tested and will compile correctly under Microsoft Visual Studio 2015 and Microsoft Visual Studio 2017; Clang 3.3, 3.4, 3.5, 3.8, and 3.9; and GCC 6.3.

For the rest of this document, when referring to C++ code, the term *class* will be used to describe any data structure. Because C++ treats the keywords *class* and *struct* the same, except everything in a *class* is private by default, it helps to have a common name to refer to.

2 Literature Review

This part of the report will analyse 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.

2.1 Runtime Reflection in Dynamic Languages

Dynamic languages, like JavaScript (Eich, 1995), Python (Rossum, 1991), and Lisp (McCarthy et all, 1958), have very powering runtime introspection features. This is due to the fact that the language is not directly compiled into native assembly code, where everything has a solid memory address that can be accessed, and a size it takes in memory. Because of this, you can change a type that was an integer into a much larger type with a simple reassignment, and the interpreter just handles it.

Python has some powerful introspection features for outputting information about a type at runtime. Figure 1 shows an example of this.

```
class TestClass:
    i = 0
    j = 0

    def __init__(self):
      pass

test_class = TestClass()
print(dir(test_class))

"""
Prints:
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'i', 'j']
"""
```

Figure 1 - A simple introspection example in Python

2.2 Runtime Reflection in strongly types languages

The language C# has some advanced and powerful reflection abilities. 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 in simple comparisons. An example of the C# *GetType* method is shown in Figure 2 which compares two integers, and then an integer and a floating pointer number.

```
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 2 - An example of C#'s GetType method and its output

C# also provides ways to retrieve the properties of a class at runtime. This is possible because each class that supports the *GetType* method also supports the *GetProperties* method. The *GetProperties* method can be used to iterate through a class, as shown in Figure 3.

```
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 3 - C#'s GetProperties method being used to get all the 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 dynamic languages but with a statically compiled language's benefits, namely syntax checking for errors. 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 to serialize objects and output their names and values. The Java beans API allows the user to analyse classes to discover properties, methods, and events. While this functionality is definitely a good thing, it has some drawbacks. Bean objects must have; a public no-argument constructor, a public get and set method 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.

2.3 Compile time introspection in other languages

Compile time introspection in languages has some significant benefits and drawbacks compared to runtime. While runtime introspection can be significantly more powerful than compile time, allowing the user to manipulate and change data at runtime, it also has a significant performance penalty compared to compile-time introspection.

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* are the data the variable stores. Using this, it provides ways to convert types to strings, for outputting. Figure 4 shows a simple example of introspection in Go; the program iterates through all of the members of a data structure. Figure 4 was tested under Go version 6.1.2.

```
package main

import "reflect"
import "fmt"

type TestStruct struct {
    a, b, c int
}

func main() {
    var test_struct interface{} = TestStruct{1, 2, 3}
```

```
    // Iterate through and print all values in the struct.
    value := reflect.ValueOf(test_struct)
    for i := 0; i < value.NumField(); i++ {
        fmt.Println(value.Field(i))
    }
    /* Prints:
        "1
         2
         3"*/

    // Iterate through and print information about each member.
    struct_type := reflect.TypeOf(test_struct)
    for i := 0; i < struct_type.NumField(); i++ {
        fmt.Printf("%+v\n", struct_type.Field(i))
    }
    /* Prints:
    {Name:a PkgPath:main Type:int Tag: Offset:0 Index:[0] Anonymous:false}
    {Name:b PkgPath:main Type:int Tag: Offset:8 Index:[1] Anonymous:false}
    {Name:c PkgPath:main Type:int Tag: Offset:16 Index:[2] Anonymous:false}*/
}
```

Figure 4 - Example of iterating through a data structure in Go.

The programming language D also provides a lot of tools for compile-time introspection. This allows it to introspect variables while avoiding the runtime costs many other languages have for using such features. However, due to this, it can be slightly more limiting than other languages.

Adam D. Ruppe (2014) discusses a powerful introspection feature; the *__traits* function, which can retrieve all the introspection information about a data structure. Using this function, which is built into the language, you can get everything in a data structure, including traits, members, methods, and virtual methods.

The examples in Figure 5 show uses of the *__traits* method to discover introspective information about a data structure.

```
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 5 - Some examples of compile-time introspection in D.

The D programming 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 and compare types. In D, type comparisons must be wrapped up in an *is* statement, which tests that the type is semantically and syntactically correct. Figure 6 shows 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 6 - Using D's typeof operator to test whether something is a function or not.

2.4 External Introspection Tools

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

Boost, which is a very popular C++ library, provides some aid for serialization. Ramey (2004) created Boost Serialization, and it allows user to turn classes into a sequence of bytes, from which the entire state of the class can be re-created. However, a major limitation of Boost Serialization is that it requires some intrusive code in order to set it up.

One of the most commonly used C++ introspection tools is the Meta Object Compiler (Oliver, 2016), which will from now on be referred to as *Moc.* Moc's popularity stems from the fact it is coupled with the popular framework Qt (Trolltech, 1991). Moc has some interesting features. One of them is the ability 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 turn calls all the functions associated with that action.

While Qt's Moc is a good tool, its design is very error prone and will mask bugs, with no compile error or runtime assert, and just silently fail. Figure 7 shows an example of where Moc works well, and where some of the errors in using it can lie.

```
#include <QObject>

// Must inherit from QObject
class Counter : public QObject {
    Q_OBJECT // Qt required macro

public:
    int value = 0;

public slots: // Qt keyword.
    void Counter::set_value(int value)
    {
        if (value != this->value) {
            this->value = value;
            emit value_changed(value);
        }
    }

Signals: // Qt keyword.
    void value_changed(int new_value);
};

int main(int argc, char **argv) {
    Counter a, b;

    // Used to connect slots and signals. We connect a to b, so that when we
    // call a.set_value, it "emits" a value changed to b.
    QObject::connect(&a, SIGNAL(value_changed(int)),
                     &b, SLOT(set_value(int)));

    // When we change a using set_value, we emit a value_changed signal to b
    // so b's value will change as well.
    a.set_value(1); // a = 1, b = 1

    // We set up a to connect with b, not b to connect with a. So when we change
    // b nothing happens to a.
    b.set_value(2); // a = 1, b = 2

    // An example where Qt is very error prone. If the function does not exist,
    // Qt just silently fails here, with no compile error or runtime assert.
    QObject::connect(&a, SIGNAL(no_function(int)),
                     &b, SLOT(no_function (int)));
};
```

Figure 7 - A sample Qt program.

Moc also drags in a lot of code, including the entire Qt framework, and keywords, which the user must understand how they work, which make maintainability much harder. It also forces the user into a very specific style of programming. The tool discussed in this paper had its API designed very carefully so that it does not require the user to change their code significantly to use it.

There are various others 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 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. The online bug list is hundreds of entries long, and some of them are years old. The tool discussed in this paper has an advantage over Qt in that regard, because the tool is very small and focused, so it is not spreading itself very thin and attempting to do too much, at a cost to the quality.

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 is a macro called *UCLASS* for classes, *UFUNCTION* for member functions, and *UPROPERTY* for member variables. Using this allows developers to introspect and generate their code in very specific and powerful 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 an application that was not a 3D package, it would not be suitable.

Another issue is that it has introduced a lot of Unreal-specific keywords in the form of macros. Having a lot of these throughout 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.

2.5 Current State of Introspection in C++

As of C++11, C++ has some limited support for introspection. This is done via the Type Traits library, and the Decltype and Auto Specifiers. The Decltype Specifier provides a way to get introspection information on a type at compile time. This is very similar to the *TypeOf* operator which *GCC* introduced as an extension into their C-language compiler.

Figure 8 shows some examples of using the Decltype Specifier in order to create types based on another type.

```
#include <iostream>

int main(int argc, char **argv) {

    int a;
    decltype(a) b;    // b is of type int.
    decltype((a)) c; // c is type int &.

    int *ptr_a;
    decltype(ptr_a) ptr_b;  // ptr_b is of type int *.
    decltype(*ptr_a) ptr_c; // ptr_c is of type int *&.

    return(0);
}
```

Figure 8 - Example showing decltype operator.

As you can see from the Figure 8, the Decltype Specifier is very powerful, but has some odd features. If we call *decltype* and put two brackets around the variable, instead of one, then the type of the variable is not a reference, and not a value. This odd behaviour can lead to some strange situations, as shown in Figure 9.

```
decltype(auto) func1() { int res = 0; return res;  } // Returns int.
decltype(auto) func2() { int res = 0; return(res); } // Returns int &.

int main(int argc, char **argv) {
    int one = func1();
    int two = func2();

    return(0);
}
```

Figure 9 - An example showing an easy mistake to make using the decltype operator.

In Figure 9, the only difference between *func1* and *func2* is whether *res* is wrapped in brackets or not. This can cause unintentional undefined behaviour, as the programmer would not have to know this part of the specification in order to avoid it.

In Figure 10, we also see another odd feature of the Decltype Specifier. If you dereference a pointer, then you get a reference to the type, not the type itself. Figure 10 shows an example where this could have been useful, had dereferencing within the Decltype Specifier returned the type, and not a reference to the type.

```
int main(int argc, char **argv) {
    // Common C idiom for allocating memory. Means that if the
    // type of a changes, the allocation is still correct. This
    // does not work in C++, however, because you have to cast
    // the void * return from malloc to the type.
    int *a = malloc(sizeof(*a));

    // This could have been a nice design idiom in C++, but the
    // decltype operator returns a reference to the type on
    // dereference, not the type.
    int *b = new decltype(*b);
```

```
    return(0);
}
```
Figure 10 - Comparison of a common memory-allocation idiom in C, and what could have been a nice C++ idiom.

C++11 also has the Type Traits library, which can get some introspection information about types and some relationship information about them. Figure 11 shows some examples within the Type Traits library.

```
#include <iostream>
#include <type_traits>

class Base {};
class Test : public Base {};

class Complex { virtual void foo() = 0; };

int main(int argc, char **argv) {
    // false
    std::cout << "Is Base the base class of Test? "
              << std::is_base_of<Test, Base>::value << std::endl;

    // true
    std::cout << "Is Base the base class of Test? " <<
                  std::is_base_of<Base, Test>::value << std::endl;

    // true
    std::cout << "Are Test and Test the same? " <<
                  std::is_same<Test, Test>::value << std::endl;

    // true
    std::cout << "Is Test a Plain Old Data type? " <<
                  std::is_pod<Test>::value    << std::endl;

    // false
    std::cout << "Is Complex a Plain Old Data type? " <<
                  std::is_pod<Complex>::value << std::endl;

    // true
    std::cout << "Is signed? " << std::is_signed<int signed>::value << std::endl;

    // false
    std::cout << "Is signed? " << std::is_signed<int unsigned>::value << std::endl;

    // false
    std::cout << "Is signed? " << std::is_signed<Test>::value << std::endl;

    return(0);
}
```
Figure 11 - Example of type_traits library.

The Type Traits library was a good step towards introspection in C++, however it is still very lacking. There is no way to convert a type into a string for sterilization, print a class to the console, or get the base type of a class. Most of the features within the C++ Type Traits library are useful as part of an introspection system, but they cannot be used to make a generic one themselves.

C++17 provides some extra introspection information, which the C++11 standard lacked, in the form of structured bindings. Figure 12 shows an example of what structured bindings can do, and how they were done before C++17.

```
#include <iostream>

class TestOne {
public:
    int i;
    bool b;
    float f;
};

int main(int argc, char **argv) {
    Test test = {10, true, 3.14f};

    auto& [ i, b, f ] = test;

    // i, b, and f are now references to the members of test.

    return(0);
}
```

Figure 12 - Structure Bindings example

Structure bindings allow for some generic code examples to work. Figure 13 shows an example of using structure bindings to print a generic class to the console.

```
#include <iostream>

class TestOne {
public:
    int i;
    bool b;
};

class TestTwo {
public:
    short s;
    float f;
};

template<typename T>void print_class(T &t) {
    auto& [a, b] = t;

    std::cout << a << ' ' << b << std::endl;
}

int main(int argc, char **argv) {
    TestOne test_one = { 1, true  };
    TestTwo test_two = { 2, 3.14f };

    print_class(test_one); // Prints "1 1"
    print_class(test_two); // Prints "2 3.14"

    return(0);
}
```

Figure 13 - Generic Structure Bindings example.

Figure 13 has some obvious, and severe, limitations. The class must have only primitives in it, or the call to *std::cout* will fail. This could be overcome by having every class require an overload of the Left Shift Operator so *std::cout* would work on it, but doing so would make the generic *print_class* function redundant, because you would have to manually print out each variable anyway.

The class must also have exactly two members, because that is how many the line copying the data structures members is expecting. Because there is no native way to get the number of members in a C++ class, this code cannot become more generic without adding some boilerplate code, like making sure each class has a member that stores the number of members as a constant expression. This is very error prone, however, as the programmer could easily miss it when modifying the class.

2.6 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 classes for the program to use 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 class are very different than what they would want from a function.

One of the issues discussed is how to introspect Unions. Unions would be very difficult to introspect, at least to the same extent as they propose introspecting classes, simply because of how limited they are in C++. It is also unclear whether Unions should generate their own meta-type or whether they should be paired together with class meta-types.

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”.

Figure 14 provides a small example, using Chochlík’s (2016) fork of Clang, where he implemented a version of the proposed reflection facilities, in order to get the number of members in a class.

```
#include <reflexpr>
#include <iostream>

class 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 14 - Example, using Chochlik's Clang fork, to get the number of members in a class.

3 Current Work

3.1 Note on section 3

The C++ specification is sometimes a little loose on terms. An example would be, from Figure 15, what is an *int* to an *int *?*

```
int integer;
int *ptr;
int &ref;
int arr[32];
```

Figure 15 - Different "types" of an int in C++.

If we dereference *ptr*, then its type becomes an *int*. However, the C++ standard does not have a well-defined term for what an *int* is to an *int **, or 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.

3.2 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 class, this tool aims to be compatible with vanilla C++ code.

The code generated from the tool requires a C++11 compile to work. The code has been tested in C++11, 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 dynamic or shared libraries, and statically links to the C Runtime Library on Windows. 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 C-Runtime Library it is linked to, and if that gets updated it may break their code.

The tool has been tested with; GCC version 4.8.4; Clang versions 3.4, 3.5, and 3.8, and Visual Studio 2015 and Visual Studio 2017. It has been tested on Windows 8, Windows 10, and Ubuntu 14.04.5.

3.3 Usage

The pre-processor is just a small command-line tool which takes some arguments. It is a 572 KB large executable, with no dependencies on any dynamic linked or shared libraries, except system ones. Also, so that people would not be discouraged from using the tool, special attention was given to make sure it could parse and output text at a very high speed.

A simple example of using the tool is provided in Figure 16.

```
preprocessor test_code.cpp
g++ test_code.cpp
```

Figure 16 - Example using the tool with GCC,

The first line from Figure 16, *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 16, *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 17 - Example using the tool, and passing in two files.

Figure 17 shows an example of using the tool with two files, *test_code_one.cpp* and *test_code_two.cpp*. Figure 17 will still generate the directory *pp_generated* and *static_generated.h*, like Figure 16, 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 enumeration, at compile time.
- Convert a string into an enumeration index at runtime or compile time.
- Convert an enumeration index into a string at runtime or compile time.

3.4 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. Figure 18 shows how you could call the tool and pass some flags in.

```
preprocessor test_code.cpp -e -t
```

Figure 18 - Example calling the tool and passing flags in.

3.5 Google Test

The Google Test framework (Google, 2016) was used in order to test the parser, and find bugs quickly. Using it allowed large changes to be performed on the codebase, while ensuring existing functionality kept working.

Figure 19 shows a small example of a test in the code, which makes sure that the number of members in a class is correct.

```
TEST(StructText, number_of_members_test) {
    char *str = "class A { int a, b, c; };"
     ClassData gen = parse_class_test(str);
     ASSERT_TRUE(gen.member_count == 3)
          << "Error: Number of members in a class was not correct.";
}
```

Figure 19 - Example of using Google Test.

First, the code creates a dummy string, which has a simple class with 3 members. Then, it passes this string into the *parse_class_test* function, which returns a *ClassData* data structure containing all the relevant information on the class parsed. Finally, it does a simple comparison to make sure the number of members parsed is actually three. An assertion would fail if the number of members was not correct here, indicating a bug in the parsing code.

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

3.6 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.

During development, an attempt was made to get rid of the custom C++ parsing code, and replace it with Clang. However, due to time constraints and the complexity of integrating Clang, this was decided against.

3.7 TypeInfo specialisation

The generated code has a special templated class called *pp::TypeInfo*. The default implementation is shown in Figure 20.

```
template<typename T> class pp::TypeInfo {
public:
    using type      = void;
    using weak_type = void;
    using base      = void;

    static constexpr char const * const name      = NULL;
    static constexpr char const * const weak_name = NULL;

    static constexpr size_t const member_count = 0;

    static constexpr bool const is_ptr = false;
    static constexpr bool const is_ref = false;

    static constexpr size_t const base_count = 0;

    static constexpr bool const is_primitive = false;
    static constexpr bool const is_class     = false;
    static constexpr bool const is_enum      = false;
};
```

Figure 20 - pp::TypeInfo's default implementation.

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 create template specializations of this class for each primitive, class, and enumeration in the file.

A subtle design decision, worth noting, is that the *type* field is set 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 source 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 be fixed.

Figure 21 shows an example of a typical class, and Figure 22 shows how the *pp::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 21 - Simple class example.

```
template<> class pp::TypeInfo<SomeClass> {
public:
    using type      = SomeClass;
    using weak_type = SomeClass;
    using base      = BaseClass;

    static constexpr char const * const name      = "SomeClass";
    static constexpr char const * const weak_name = "SomeClass";

    static constexpr size_t const member_count = 3;

    static constexpr bool const is_ptr       = false;
    static constexpr bool const is_ref       = false;

    static constexpr size_t const base_count = 1;

    static constexpr bool const is_primitive = false;
    static constexpr bool const is_class     = true;
    static constexpr bool const is_enum      = false;
};
```

Figure 22 - Template specialization of pp::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 23, but, for the sake of brevity, only the first line is shown. Each of the relevant fields, such as *is_ptr* and *is_ref*, are changed depending on 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 23 - Six generated specializations of pp::TypeInfo for SomeClass.

The first field of *pp::TypeInfo* is just an alias which is set 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 24 shows an example of using *pp::TypeInfo::weak_type*. The nice thing about this example is how robust it is. If the variable *i* is changed from a pointer to an integer to an integer, 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;      // j is an int *.
    pp::TypeInfo<decltype(i)>::weak_type k; // k is an int.

    return(0);
}
```

Figure 24 - pp::TypeInfo::weak_type example

The next two fields declared in *pp::TypeInfo* are *name* and *weak_name*. Like *type* and *weak_type*, these correspond 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 25.

```
#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 25 - Outputting type and weak_type.

The field *is_ptr* on the TypeInfo class is just a boolean to tell if something is a pointer. Figure 26 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 26 - pp::TypeInfo::is_ptr example

Another example of when it would be useful to test if something is a pointer would be in a template function that can take a pointer or a value. Because, in C++, members of a class pointer must be dereferenced and then accessed, using the arrow operator, while normal classes can only be accessed using the dot operator, this can prove problematic if the user wishes to accept either.

Figure 27 shows an example that helps solve this problem, using C++17 constant expression if statement. Because *is_ptr* is a compile-time constant, it can be used with the constant if statements. In fact, every member of the *pp::TypeInfo* class is a constant expression, so you could do a constant if 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) {
        // If var is a pointer, then it will enter this code path and the other
        // one will not even be compiled. If it's not a pointer, then it will enter
        // the other one and this one won't be compiled. This can be very useful.
        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 27 - C++17 example, using pp::TypeInfo::is_ptr.

The *base_count* fields of *pp::TypeInfo* is just an integer which tells the user how many classes the class templated on inherits from. Figure 28 shows a basic example of using it.

```
#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."

    std::cout << pp::TypeInfo<BaseOne>::name << " inherits from " <<
                 pp::TypeInfo<BaseOne>::base_count << " classes.";
    // "BaseOne inherits from 0 classes."


    return(0);
}
```

Figure 28 - pp::TypeInfo::base_count example.

The example in Figure 28 could be taken further, in order to develop a generic function which can print how many classes any class passed into it inherits from. Figure 29 shows this.

```
#include "pp_generated/test_code_generated.h"
#include <iostream>

class BaseOne {};
class BaseTwo {};
class BaseThree {};

class Test : public BaseOne, public BaseTwo, public BaseThree {};

// Generic function to print how many classes any type inherits from.
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;
    int test3;

    print_base_class_count(test);  // "Test inherits from 3 classes."
    print_base_class_count(test2); // "BaseOne inherits from 0 classes."
    print_base_class_count(test3); // "int inherits from 0 classes."


    return(0);
}
```

Figure 29 - Generic pp::TypeInfo::base_count example.

The final field of the *pp::TypeInfo* class 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, due to issues like the "diamond problem" (Milea, 2011). And due to some limitations in C++, it was decided that only the first class inherited from will be available. In the future, this may be expanded upon.

Figure 30 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;

    // Create an instance of test's base class.
    pp::TypeInfo<decltype(test)>::base test_base;

    // Prints "Base".
    std::cout << pp::TypeInfo<decltype(test_base)>::name << std::endl;

    return(0);
}
```

Figure 30 - pp::TypeInfo::base example.

The example in Figure 30 can be done without even creating an instance of the base class. This is shown below in Figure 31.

```
#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 31 - pp::TypeInfo::base example.

Using the *base* field of *pp::TypeInfo* within itself, you can go up a hierarchy of inherited classes to get the one at the top. Figure 32 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 32 - Complicated pp::TypeInfo::base example.

Making the example in Figure 32 even more generic, we can write a function that will find the highest-level base class of any type and print it. Figure 33 demonstrates this. It will keep recursively generating instances of the function *hierarchy*, until the top-level version is found. We know we've reached the top level because the *base* type will be *void*.

```
#include "pp_generated/test_code_generated.h"
#include <iostream>

class BaseTwo {};
class BaseOne : public BaseTwo {};

class Test : public BaseOne {};

// Generic function, which will go up any inheritance hierarchy and will print
// out the top-most member.
```

```
template<typename T>void hierarchy() {
    if(pp::type_compare(pp::TypeInfo<T>::base, void)) {
        std::cout << "The most base classes type is " <<
                    pp::TypeInfo<T>::name << std::endl;
    } else {
        hierarchy<pp::TypeInfo<T>::base>();
    }
}

// Need to specialize for “void”, so we know when we’ve reached the top of
// the inheritance hierarchy.
template<>void chain<void>() {}

int main(int argc, char **argv) {
    hierarchy<Test>();    // Prints “BaseTwo”.
    hierarchy<BaseOne>(); // Prints “BaseTwo”.
    hierarchy<BaseTwo>(); // Prints “BaseTwo”.

    return(0);
}
```

Figure 33 - Generic pp::TypeInfo::base example.

A lot of the code for *pp*::*TypeInfo* can also work hand-in-hand with the Type Traits library. Figure 34 shows an example of getting the base class using *pp::TypeInfo*, but statically asserting it using the Type Trait library’s *std::is_base_of* function.

```
#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 34 - Mixing type_traits and pp::TypeInfo::base.

3.8 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 35.

```
MEMBER_TYPE * pp::get_member(CLASS_TYPE *variable, size_t index);
```

Figure 35 - Definition of pp::get_member.

Due to some limitations on how types can work in C++, the code to access members by index is a little verbose. In C++, it is illegal to overload a function on the return type alone. Because of this, it can make getting the return type of a function difficult. The code in Figure 36, which will not compile, shows how this could be done in some languages where type information is provided at runtime.

```cpp
#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 36 - C++ equivalent of how type information is used in other languages. 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 37 starts off with some basic examples of *pp::get_member*. The function *pp::get_member* is specialised for each class in the project, and then again for each member of a class. Because of this, the return type will be different, depending on the index passed in.

```cpp
#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 37 - Basic pp::get_member example.

The function *pp::get_member* will also work for pointers. In that case, the return type will be a pointer to that pointer. For instance, if a class had a member that was a pointer to an integer, then *pp::get_member* would return a pointer to that member. Figure 38 shows an example of this, as well an example of a class within a class, and getting a member which is a *std::vector*.

```
#include "pp_generated/test_code_generated.h"
#include <iostream>

class V2 {
public:
    int x, y;
};
class Test {
public:
    int *i;
    V2 v;
    std::vector<float> float_vec;
};

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);

    std::cout << test.v.x << ' '  << *x; // Prints "2 2".
    std::cout << test.v.y << ' '  << *y; // Prints "4 4".

    // Example where the member is a vector.
    test.float_vec.push_back(0.25f);
    test.float_vec.push_back(0.50f);
    test.float_vec.push_back(0.75f);
    test.float_vec.push_back(1.00f);

    // Get a reference to the member.
    auto &vec = *get_member(&test, 2);
    for(size_t i = 0; (i < vec.size()); ++i) {
        std::cout << vec[i] << ' '; // Prints: "0.25 0.5 0.75 1".
    }
```

```
    return(0);
}
```

Figure 38 - Complex pp::get_member example.

While Figure 38 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 39 shows how this can be done using the tool, 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 39 - 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 negative one, at which point the specialisation of *print_var* for negative one 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 constant if statements, then you can change this code to work for a class within a class. Figure 40, which will only compile under a C++17 compliant compiler, like Clang 3.9, shows this.

```
#include "pp_generated/test_code_generated.h"
#include <iostream>

class V2 {
public:
    int x, y;
};

class Test {
public:
    int i; V2 v;
};

template<typename T, int index>void print_var(T *var) {
    // If we have C++ 17 compile, then we don't need the boilerplate print_var
    // specialization for when index < 0. Can just do a constexpr if to make sure
    // the member count is >= 0.
    if constexpr(index >= 0) {
        print_var<T, index - 1>(var);
        auto member = pp::get_member(var, index);

        if constexpr(pp::TypeInfo<decltype(member)>::is_primitive) {
            /* 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 {
            print_var<typename pp::TypeInfo<decltype(member)>::weak_type,
                      pp::TypeInfo<decltype(member)>::member_count - 1>(member);
        }
    }
}

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);
    print_var<Test, pp::TypeInfo<Test>::member_count - 1>(&test);


    return(0);
}
```

Figure 40 – C++17 only. Generic serialization example that supports classes within classes.

Figure 41 demonstrates how this can be used to work with pointers as well. Like Figure 40, however, this requires constant 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) {
    if constexpr(index >= 0) {
        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;
        }
    }
}
```

```
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);

    /* Prints
        "10
         5" */

    return(0);
}
```

Figure 41 – C++17 only. Generic serialization example that includes pointers.

Finally, Figure 42 combines both Figure 40 and Figure 41 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 V2 {
public:
    int x, y;
};

class Test {
public:
    int i;
    int *i_ptr;

    V2 v;
    V2 *v_ptr;
};

template<typename T, int index>void print_var(T *var) {
    if constexpr(index >= 0) {
        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 that 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) {
```

```
                print_var<
                    typename pp::TypeInfo<decltype(member)>::weak_type,
                    pp::TypeInfo<decltype(member)>::member_count - 1
                >(*member);
            } else {
                print_var<typename pp::TypeInfo<decltype(member)>::weak_type,
                          pp::TypeInfo<decltype(member)>::member_count - 1>(member);
            }
        }
    }
}

// Simple utility function to make calling print_var nicer.
template<typename T> void my_print(T *var) {
    // Has to be member_count - 1, because of zero indexing. If a class has 3
    // members, then it actually has members 0 - 2.
    print_var<T, pp::TypeInfo<T>::member_count - 1>(var);
}

int main(int argc, char **argv) {
    Test test;

    test.i = 10;

    test.i_ptr = new int;
    *test.i_ptr = 5;

    test.v = {2, 4};

    test.v_ptr = new V2;
    *test.v_ptr = {5, 10};

    my_print(&test);

    /* Prints
        "int 10
         int *5
         int 2
         int 4
         int 5
         int 10"*/

    return(0);
}
```

Figure 42 - C++17 only. Fully generic serialization example, which supports classes within classes and pointers.

The nice thing about this example is all the serialization is in user code, not behind a black box, like *pp::print* function, which is discussed in section 3.10.

Building on the previous examples even more, Figure 43 writes a class to disk. It writes a class to disk in XML format (Bray et all, 1996), which could be read by other tools in order to draw some data about the class. Figure 44 shows the output from Figure 43.

```
#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;;
}

template<>void serialize_var<Test, -1>(Test *t, std::stringstream &buffer) {}

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 43 - Serialize a class to an XML file.

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

Figure 44 – XML output from Figure 43.

In object-oriented design, it is common to set member variables to *private*, and have functions which can access and set the variable. This is to aid with encapsulation, and prevent programmers modifying data by accident.

In an object-orientated codebase, you may only want to serialise only the public members of a class. The tool lets you query, at compile time, whether a function at an index is *public*, *private*, or *protected*, using the function *pp::get_access_at_index*. Figure 45 shows the return type of this function, which is an enumeration defined in *static_generated.h*, and Figure 46 shows the definition of this function.

```
enum pp::Access {
    Access_public,
    Access_private,
    Access_protected,
};
```

Figure 45 - Enumeration returned from pp::get_access_at_index.

```
template<typename T, int index> constexpr pp::Access pp::get_access_at_index();
```

Figure 46 - pp::get_access_at_index definition.

A design decision was made to have the index in *pp::get_access_at_index* be required at compile-time, rather than calculated at run time. While having it at runtime would be more flexible, as the user could query the access rights of a member at an index, where the index is calculated at runtime, it is also slower. Having it be required at compile-time, means that the specialized function can be a constant-expression function, making it much more efficient. The efficiency over flexibility was chosen because, when the user is querying about a variable at an index, they often have the index as a constant anyway, because accessing it through *pp::get_member* requires it to be an index.

Figure 47 shows a simple example, based on Figure 39, which iterates through all the members of a simple class, and outputs them to the console. Figure 47, however, only outputs members that have public access.

```
#include "pp_generated/test_code_generated.h"
#include <iostream>

class Test {
private:
    short s; double d;

public:
    int i; float f;

    // Public "setter" methods for s and d.
    void set_s(short s)  { this->s = s; }
    void set_d(double d) { this->d = d; }
};

template<typename T, int index>void print_var(T *var) {
    print_var<T, index - 1>(var);

    // Only print variables that have their access set to "public".
```

```
    if(pp::get_access_at_index<T, index>() == pp::Access_public) {
        auto member = pp::get_member(var, index);

        char const *type_as_str = pp::TypeInfo<decltype(member)>::weak_name;

        std::cout << type_as_str << ' ' << *member << std::endl;
    }
}

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.f = 3.14f;
    test.set_s(4);
    test.set_d(3.1415f);

    /* Prints "int 10
               float 3.14"
       It completely ignores s and d, which are set to "private". */
    my_print(&test);

    return(0);
}
```

Figure 47 - Output public members of a class.

3.9 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 is that C++ forbids the comparison of types. Figure 48 shows a line that will not compile under any standard-compliant C++ compiler.

```
if(int == int)
```

Figure 48 - Invalid type comparison.

C++11 goes some way to fixing this, by providing a mechanism in the Type Traits library, *std::is_same*, which allows you to compare types. Figure 49 shows some examples using *std::is_same*.

```
#include <iostream>
#include <type_traits>

class Test {};

int main(int argc, char **argv) {
    std::cout << std::is_same<int, int>::value   << std::endl; // true
    std::cout << std::is_same<int, int *>::value << std::endl; // false
    std::cout << std::is_same<int, float>::value << std::endl; // false
    std::cout << std::is_same<int, Test>::value  << std::endl; // false
```

```
    std::cout << std::is_same<Test, Test>::value << std::endl; // true

    return(0);
}
```

Figure 49 - std::is_same examples.

The metaprogramming tool provides an API which is the same as *std::is_same* for comparing types. Its definition is shown in Figure 50.

```
bool pp::type_compare(TYPE a, TYPE b);
```

Figure 50 - pp::type_compare definition.

The reason for this duplication, is because there is no part of the generated code that requires any of the C++ Standard Template Library, although it in no way discourages it. In order to keep the tool lightweight, the decision was made to duplicate this small part, for type comparisons, rather than force the user to include the entire Type Traits library. Including the library would increase compiles times for users that did not want to use it, due to heavy use of templates (Dawson, 2014). The rest of the examples in this document use *pp::type_compare* for type comparisons, but could just as easily have used *std::is_same*.

Figure 51 shows an example of using *pp::type_compare* with *pp::weak_type* to test whether an integer and a pointer to an integer are the same.

```
#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 51 - pp::type_compare example.

3.10 Printing Classes

One of the most powerful methods available inside the tool is used for printing a class to the console. The function definition is provided in Figure 52.

```
void pp::print(TYPE v, char *buffer = NULL, size_t buffer_size = 0);
```

Figure 52 - pp::print 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 class itself.

Figure 53 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 53 - pp::print example.

The function *pp::print* goes through all the members of the class and prints them. In Figure 53, it first goes to the member *text* and prints that, then it prints the member *length*.

Figure 54 shows a more complex example, and how the *pp::print* function 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 54 - Complex pp::print example.

In Figure 54, you can see how the *pp::print* method handles different types. The first two types, which are primitives, are printed to the console normally. The third type, which is a class, has its type printed, then the serialise function recursively calls itself and prints out the members.

For the first pointer in Figure 54, *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 its members.

The first array in Figure 54, *int i_arr[2]*, is initialised within a *for* loop. Inside the function, its 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*. Its definition is shown in Figure 55.

```
size_t pp::serialize(TYPE var, char *buffer, size_t buffer_size);
```

Figure 55 - pp::serialize definition.

The function *pp::serialize* will fill out the buffer passed in 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. It requires the user to pass in the size of the buffer, to make sure it does not write to invalid memory and cause a crash. It then returns the number of bytes it actually did write, in case the user wants to do something with this information.

Figure 56, which will only compile under Microsoft Visual Studio on Windows, shows when it could be useful to write a serialized class to disk. The output is shown in Figure 57.

```
#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;
        }

        // Write to every value in the array.
        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];

        // Serialize the class "test" into the buffer variable.
        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 56 - Windows only. pp::serialize example using SEH 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 57 - Output from Figure 56.

Figure 56 shows a good example of writing class information to disk. In Figure 56, there is a class that has an array of pointers to integers. Intentionally, for the purpose of the demo, when the pointer's 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 57, it is obvious that the bug is because the sixteenth element is NULL.

While not every bug would be as obvious to see as the example in Figure 56, 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 files usually generated when something crashed.

While the function *pp::print* could have been left out, forcing users to always implement their own serialisation code, it was important to leave it in. This is because, when a programmer is using an external API, they will want easy results at the start of a project, and more control towards the end (Muratori, 2004). Having the functions *pp::print* and *pp::serialize* allows programmers that only want a class serialized quickly to be able to use the tool comfortably. Programmers that want more control over how their data is outputted can use the *pp::get_members* function.

3.11 Enumerations

Enumerations, defined under some limitations, which are discussed later, can use *pp::TypeInfo* in order to get some information about themselves. Figure 58 shows an example of using a C++11 enumeration 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);
}
```

Figure 58 - pp::TypeInfo with an enumeration.

There are also two functions defined in the API, which are unique to enumerations. Both of these function definitions are shown in Figure 59.

```
template<typename T> constexpr char const *pp::enum_to_string(T element);
```

```
template<typename T> constexpr T pp::string_to_enum(char const *str);
```

Figure 59 - pp::enum_to_string and pp::string_to_enum definitions.

Both of these functions will be calculated at compile time, where possible, because they are marked as constant expressions. The generated code will specialise each of these for each class in the project. Figure 60 shows some examples of *pp::string_to_enum*, and Figure 61 shows some example *pp::enum_to_string.*

```
#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 60 - pp::string_to_enum examples.

```
#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 61 - pp::enum_to_string examples.

The enumeration introspection data will work with both normal enumerations and C++11 enumeration classes. However, there is one important limitation; it will not work with enumerations that have not had their storage type explicitly defined. Figure 62 demonstrates the difference.

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

Figure 62 - Supported and un-supported enumerations.

The reason that C-style enumerations are not supported is that they cannot be forward declared. Until the draft *Forward declaration of enumerations* (Barbati, 2008), was accepted into the C++11 specification, there was no way to forward declare an enumeration in C++. C++11 allows forward declared enumeration if the underlying type is specified. Hence the tool, which requires forward-declarations to work, is not compatible with enumerations which do not have an underlying type.

3.12 Performance

Due to the nature of how C++ is usually compiled directly to assembly, having high-performance libraries and tool is a large concern for C++ programmers. Because of this, performance was a large concern when designing the introspection tool. Two examples are given, in Figure 63 and Figure 64, respectively, which demonstrates the high performance of the tool. On the left hand side of each figure is the C++ code, and on the right hand side is the x86 Assembly (Intel, AMD, 1978), generated by Microsoft Visual Studio 2015.

```
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

; return 0
 xor         eax,eax
```

Figure 63 - Sample program and x86 Assembly generated.

```
class SomeClass {
public:
    int a;
    int b;
    int c;
    int d;
};

int main(int argc, char **argv) {
```

```
; main
```

	push ebp mov ebp,esp
printf("The number of members in" "%s are %d", pp::TypeInfo<SomeClass>::name, pp::TypeInfo<SomeClass>::member_count);	; printf part push 4 ; member count push 171F94h ; class name push 171FA8h ; format string call 0E3C3Dh ; call printf
return(0); }	; return 0 xor eax,eax

Figure 64 - Sample program using introspection tool and x86 assembly generated.

As you can see, from Figure 63 and Figure 64, the assembly generated is identical in terms of functionality. A nice benefit of having all the members of *pp::TypeInfo* marked as constant expressions is that the assembly generated for them is almost identical to the non-introspection version. This means that, for a large number of cases, there is no runtime performance penalty for using the tool compare to traditional methods.

4 Future Work

4.1 Further C++ support

While the tool currently supports a large subsection of the C++ programming language, it is not complete. 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.

4.2 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. The reason class and enumeration introspection was prioritised over function introspection is because it is much more useful to be able to iterate through the members of a class, rather than through the parameters of a function. Function introspection will be the next large feature tackled in the system.

4.3 Error Handling

Right now, a syntax error in normal code may generate a syntax error in the generated code. While the tool does combat some simple errors, like if it sees you've inherited from a class that doesn't exist, it does not do anything with this information. These errors could be written directly to the Standard Error Stream.

4.4 Standard Template Library Support

Currently, the tool only has limited support for C++ Standard Template Library types, especially when it comes to serializing them through *std::print*. It currently supports; *std::vector*, *std::list*, *std::forward_list*, and *std::deque*. In the future, it will be able to support any of the types in the standard library.

4.5 User defined containers

In C++, it is possible to create a custom container which can be iterated through using C++11 range-based for loops. All that is required of the container is to have two member functions to get the beginning element and end element. This is demonstrated in Figure 65.

```
#include <iostream>

template<typename T>
class MyArray {
public:
    T *data;
    size_t size;

    MyArray(size_t size) {
        this->size = size;
        this->data = new T[size];
    }

    ~MyArray() { delete this->data; }
```

```
    // Required for C++11 range-based for loops
    T *begin() { return(data);        }
    T *end()   { return(data + size); }
};

int main(int argc, char **argv) {
    MyArray<int> arr(4);

    // Set every value in the array to 10.
    for(auto &iter : arr) iter = 10;

    // Prints "10 10 10 10".
    for(auto &iter : arr) std::cout << iter << ' ';

    return(0);
}
```

Figure 65 - Range-based for loop using custom container.

In order to support these custom containers, and print them correctly, the tool would just have to note which classes have defined the *begin* and *end* member functions. Then it could output the serialization code for them identical to how it outputs it for Standard Template Library types.

5 Conclusion

5.1 Abstract Introspection Problems

A lot of problems relating to introspection became very obvious during the development of this project. One of the biggest was the cost to readability when using a complicated introspection system, especially in strong-typed languages like C++, where the type of a variable cannot be mutated at runtime. Because of this limitation, some additional boilerplate must be created around the introspection system in order to make it fully generic. This presents something of a problem, because the idea behind the introspection is to remove boilerplate code. It can get worse still, because if the boilerplate to set up introspection is more complex than the boilerplate to just serialize each class separately, then the case for introspection as a real tool, rather than a novelty, is much weaker.

5.2 Introspection Problems Specific to C++

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, do not depend on the order of compilation, and have a module system for including files, which means the introspection data is gathered 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 class. An example would be, if a class has another class 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 to 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.

5.3 Final Thoughts

Overall, the project proved successful in providing a clean API for programmers to access introspection data in C++. The project can definitely be expanded upon in the future to support more of the C++ language. The research provided in this document could also serve as a strong starting point for providing complicated introspection features in a compiled language through a clean API.

6 References

American National Standards Institute (2016). *C++ Official Standard.* Available from: http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS/ISO/IEC+14882:2014+(2016)

Barbati A. (2008). *Forward declaration of enumerations (rev. 3).* Available at: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2764.pdf

Beck K., Beedle M., Bennekum A., Cockburn A., Ward Cunningham., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffies R., Kern J., Marick B., Martin R., Mellor S., Schwaber K., Sutherland J., Thomas D. (2001). *Manifesto for Agile Software Development.*

Bray T, Paoli J, Sperberg-McQueen C, Maler E, Yergeau F., Cowan J. (1995). *Extensible Markup Language, XML*

Bright W. (2001). *D programming language.*

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/

CollabNet. (2000). *Apache Subversion revision control system.*

Dawson B. (2014). *Making Compiles Slow Through Abuse of Templates.* Available at: https://randomascii.wordpress.com/2014/03/10/making-compiles-slow/

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

Eich B. (1995). *JavaScript programming language*.

GCC XML. (2015). *GCC_XML.* Available at: *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/

Google. (2016). *Google Test Framework*. Available at: https://github.com/google/googletest

Gosling J. (1995). *Java programming language*.

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

Hoare G. (2010). *Rust programming language.*

Intel, AMD (1978). *x86 instruction set architecture*.

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

McCarthy J., Russell S., Hart T., Levin M. (1958). List.

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

Microsoft. (2000). *C Sharp programming language*.

Microsoft. (1995). *Windows NT family of operating systems.*

Milea A. (2011). *Solving the Diamond Problem with Virtual Inheritance.* Available at: http://www.cprogramming.com/tutorial/virtual_inheritance.html

Muratori C. (2004). *Designing and Evaluating Reusable Components.* 8.19 mins. [Online] Available*:* https://www.youtube.com/watch?v=ZQ5_u8Lgvyk

Oliver G. (2016). *Qt's Moc*.

Perforce Software. (1995). *Perforce Helix version control system.*

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

Rossum, G (1991). *Python*.

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/

Torvalds L. (1991). *Linux operating system.*

Trolltech. (1991). Qt.

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.

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: 4

Date/Time: 19/01/2017 - 2:00pm

PROGRESS

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

- Finish interim report.
- Investigated replacing parser with Clang.
- General code tidying.

AGENDA FOR FORMAL MEETING (Example)

- Discuss interim report.
- Discuss what is still to be done.
- 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: 4

Date/Time: 19/01/2017 - 2:00pm

MINUTES

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

- Attempt to replace the custom parser with Clang.
- Issues with building Clang on windows.

PLAN

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

For the next month:

- Attempt to get Clang building from source.
- Handle any bugs that come up.

Beyond the next month

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

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: 5

Date/Time: 02/02/2017 - 10:00am

PROGRESS

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

- Investigated replacing parser with Clang.
- Basic support for standard template library.

AGENDA FOR FORMAL MEETING (Example)

- Discuss interim report.
- Discuss the presentation.
- Further discuss Clang as a parser.
- 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: 5

Date/Time: 02/02/2017 - 10:00am

MINUTES

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

- Discussion on replacing Clang as the parser, and issues relating to that.
- How to structure the presentation.
- How to generalize some of the code more to work with other C++ STL containers.

PLAN

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

For the next month:

- Create a presentation which discusses the project and have a small demo ready.
- Generalize the code to work with 2 STL containers.
- Continue working with Clang to verify if replacing the parser with it is an option.

Beyond the next month

- Support more C++ features for introspection.
- Finish the full 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: 6

Date/Time: 09/03/2017 - 3:00pm

PROGRESS

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

- Template specialization of introspection data
- Work on report

AGENDA FOR FORMAL MEETING (Example)

- Discussion of current approach
- What should be in the report

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: 6

Date/Time: 09/03/2017 - 3:00pm

MINUTES

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

- Discussion on last-minute stuff in project.
- How to structure report.

PLAN

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

For the next month:

- Finish report.