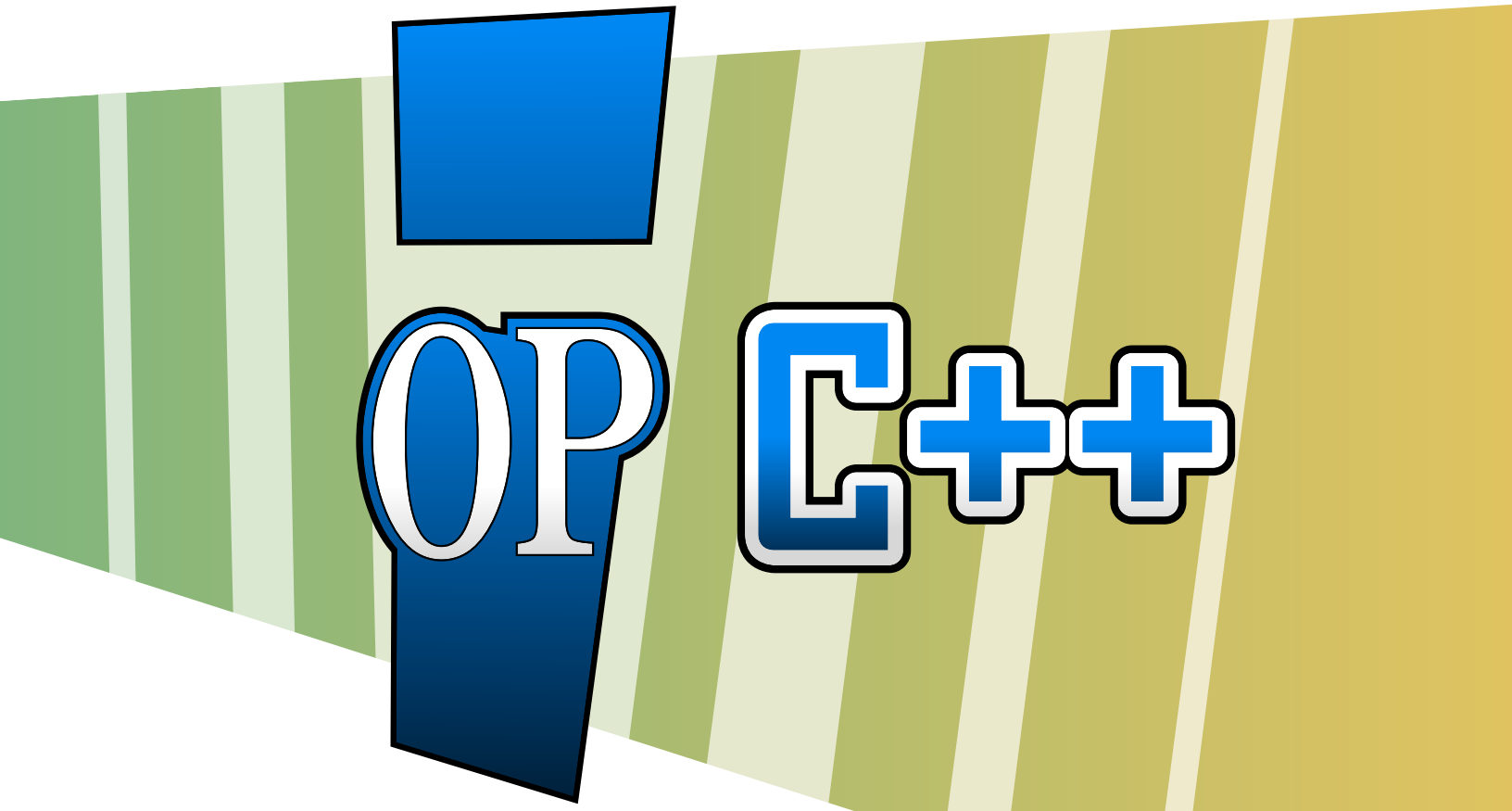

opC++™ Compiler Manual

Version 1.0

Contents

1: Introduction

1.1 What is opC++?

opC++ is a tool and language for adding additional language features to C++.

1.2 Design

Language Design

Tool Design

1.3 Manual

2: Concepts

note: the challenge here is to get the reader (and us) to think of these things not in terms of the implementation, but in terms of concepts and new ways of thinking about code.
note: I think a good way of looking at opC++ concepts, is a widening of definitions to enable capturing features from other languages. Then look at it as giving you the ability to constrain these wide definitions into usable and specific constructs. Yes this is what the approach should read like - I think we have a tendency to say that it captures all features - but it just enables capturing some features, and it will improve as the tool evolves.

2.1 Overview

This chapter outlines the concepts found in opC++. The first concept is a "category," a more general version of classes in OOP. Second, opC++ adds an "enumeration" concept, which is a more general version of an enum. Third, opC++ adds a modifier concept - which is a compile time version of attributes/annotations found in other languages. Last we look at the ability in opC++ to define language constraints.

2.2 Category

A Category is an OOP class which captures any number of language features. We'll look at the possible features a category can capture in this section, and what useful features currently exist in other languages. Specifically we'll look at what "class" means in C#, C++, Java, and UnrealScript. Each of these languages target different uses - from the lower level uses of C++ to the game-specific language Unrealscript.
Diagram of C#, C++, Java, Unrealscript and a metaclass - the category.

What are language features? Language features add functionality at a built-in language level instead of relying solely on collections of code. Features can make programming much easier - especially when specially designed for a purpose. Often a language feature is something that was very difficult to implement in a previous language. These features do not come without costs however - if you add a feature at a class level, and it remains unused, the feature can impose an unnecessary cost on the program. Due to these costs languages like C++ have almost no hidden costs, but have fewer "nice" features.
The features of a class are usually entirely specific to the language the class is implemented in. When we describe features we usually refer to a language specifically - such as that 'C# has attributes' or 'Java has reflection'. With Categories we can pull the feature out of the language context and apply it instead to a particular named category. We can state that 'the class category is private by default' or that 'the windowclass category supports windowing' - this lets us talk about features independent of language.

Now we'll look at some of the features in common general purpose languages, and how they mesh with categorization. We'll also examine a language with a more specific purpose, the game scripting language UnrealScript. Looking at this range of languages is important to get a feel for how languages are becoming more project specific - we can see this with attributes in general purpose languages, and with custom project specific languages becoming more common.
The current trend has been to move from general purpose languages to more specific languages, with language features becoming more project focused.

C++ class C++ provides a no frills definition of a class. It's defined as an object oriented programming construct that encapsulates functions, data, and can inherit its type, functions and data from other classes. C++ actually provides two types of classes, "class" and "struct". The differences are technically confined to their default access specifiers. In practice you use class when working with heavy objects and struct when creating lightweight data structures. We can view class and struct as categories with very minor differences.
class and struct as categories.
table showing differences.
Other features in C++ classes include defining functions as polymorphic (virtual), abstract/pure-virtual, and defining them as inline. C++ also gives good control over data layout, data sizes, and on certain compilers, data byte alignments - all useful when you need to work closely with hardware.
Example showing polymorphism, declaring functions and data.
C++ is a compile-time oriented language, so there are few dynamic typing features. The one dynamic typing feature standard in the language, Run Time Type Information (RTTI), is considered to be slow in practice, and is only available if a type is polymorphic (has virtual functions).
Example showing C++'s dynamic features.

C# class Since C# is a Just-in-time compiled language (JIT), it can do more things with its programs on demand, versus a compiled language like C++. C# classes support a number of useful features, including attribute oriented programming, full reflection support, and some valuable new types. Attribute oriented programming has been proven useful as it keeps any data relevant to a member written at the declaration location. Attributes can be used to flag particular members, add extra data to a specific member (metadata) and can be extremely valuable when dealing with unknown classes in a library framework.
Example of attribute oriented programming in C#.
Other design changes were made when adapting C++ like functionality into Microsofts .NET Framework. C# only supports single inheritance of classes - and instead provides an alternative construct, made popular in java, the "interface". In general most things that are done as multiple inheritance in C++, can be done with a meaning closer to the actual intent with interfaces. Practically, using interfaces means that you can attach behaviors to unrelated types, and discover whether a type implements a behavior using the interface casting mechanism in C#.
Example of interfaces in C#, example of casting to an interface to discover implemented behavior.
C# also adds the delegate type to the language. Delegates are a more general version of the function pointer in C++, with much nicer syntax, and the ability to do multiple dispatch.
Example of delegate use in C# vs function pointers in C++.
Table of important features in C# classes. - run-time attributes - interfaces - delegate

Java class Since Java is an interpretted language, it's comparatively easy to change code after the code has been compiled. This has made Java a popular language for experimenting with new language features - such as Aspect Oriented Programming (AspectJ). As Java has evolved it has added support for attribute oriented programming, as well as code generation based on annotations. Using the standard "Metadata Facility for Java" or the XDoclet project, you can generate code as a byproduct of using attributes.
Example of attributes and their possible implications in Java.
Java adds a special keyword only for serialization, "transient". This keyword tells the serialization runtime that a data member should not be serialized.
Table of important java class features - attributes - serialization support (transient) - code generation from attributes - hash - to string

Unrealscript class A common language in game development is the Unreal Engine scripting language "Unrealscript". Unlike a general programming language in a custom scripting language you always have to deal with the interface between your scripts and main application. Because of this boundary it uses keywords like "event" for native-to-script transitions, and "native" for script-to-native communication. Since it has a specific goal in mind - gameplay programming, it also contains specific keywords that only interact with the engine. The "exec" keyword makes a function callable from the engine input console. It contains class attributes - such as "ini("inifile")" which alter where settings are stored. It contains support for editor reflection on particular data members with another attribute syntax.
Example of an unrealscript class w/ attributes.
Unrealscript also deals with storing off default data values, which may be data or objects. This default value data can be read statically by script code.
Example of default instances.
As a game language UnrealScript also pays particular attention to AI and time. Unrealscript has support for sequenced execution of actions (latent functions), and state based switching of functions.
Example of states in Unrealscript.
Unrealscript is an example of a language written specifically for one project. It's feature rich because of this, and although each feature can add a size or performance cost to the language, they make the language much better.

Abstract class now generalize into a definition for a category. look at what features each language brings to the table. give example categories that might be useful to have.
From the previous examples, we've seen that languages can differ greatly in terms of language features. We see that as a language gets more specific, it can add more features without unnecessary costs. Also, the language of a particular class determines its features.
Table summarizing these languages and their features
In opC++ we'll be dealing with a more abstract class called "categories" which can group language features beyond hardcoded class and struct identifiers.

2.2.1 Declaring Members

what kind of members can you declare in categories? this should maybe do an overview of c++ even.
this should have a table showing all the syntax valid within opC++ and C++

this should have a table showing all syntax valid within C++ only
this should have a table showing all syntax valid within opC++ only

2.2.2 initialization

In opC++, you can initialize data members directly at their definitions. This is a looser syntax than standard C++, which can only initialize const static integer data members. The result of the initialization syntax can be defined as a language feature - it does not coorespond directly to default constructors, although it can be made to work that way.
the loose initialization rules in opcpp.

2.2.3 syntax

include a table detailing valid and invalid syntax in opcpp vs c++.
this should display a table with valid/invalid non-member syntax.

Allowed Category Preprocessors
#if
#ifdef
#ifndef
#elif
#else
#endif

Table 2.1: Table showing which C++ preprocessors are allowed inside categories.

2.3 Enumeration

what is an enum?
why is it useful to abstract enum into Enumeration types?
example of a generated strongly typed Enumeration.

2.4 Modifier

Modifiers in opC++ are similar to attributes and annotations in other languages. However, modifiers are a language level feature, and may observe special rules at compile time.
Modifiers are enabled for specific categories, and can be defined for data members, function members, and category definitions.
example of modifiers in opC++

2.4.1 basic modifiers

what does a basic modifier look like?
on data members? on functions?
example of basic modifiers in opC++

2.4.2 value modifiers

what does a value modifier look like?
on data members? on functions?
example of value modifiers in opC++

2.4.3 data modifiers

const, volatile
modifiers can only apply to data, functions or both
examples of data modifiers (as abstract attributes)

2.4.4 function modifiers

explicit, virtual, inline
modifiers can only apply to data, functions or both
examples of function modifiers (as abstract attributes)

2.4.5 access modifiers

public, private, protected

2.4.6 special modifiers

uninline, inline, static

2.4.7 automatic modifiers

In addition to specified modifiers, opC++ also automatically applies a number of modifiers to declarations. These modifiers exist in order to allow you to filter through and constrain language syntax. Currently these modifiers only apply to the original statement and do not derive from types declared elsewhere.
example of a data statement and the generated modifiers example of a function statement and the generated modifiers
example of automatic modifiers only applying to the exact statement.
need a whole table describing what modifiers are generated when

2.5 Constraints

Constraints in opC++ allow you to specify extra language options for validation. Disallow constraints let you specify invalid modifier combinations. Modifier Value constraints allow you to specify what values are valid for a modifier.

2.5.1 disallow

Disallow constraints may specify a custom error message.
Example: tighten back the initialization options.
Example: dont allow certain modifier combinations
Example: enforce naming conventions.

2.5.2 modifier values

Modifier value constraints may specify a custom error message.
Example: only want float numbers in a version() modifier.
Example: only want an identifier in another modifier.

3: opC++ Standard Dialect

3.1 Overview

The opC++ Standard Dialect is a dialect of C++ that adds data reflection, data member metadata, type reflection, full serialization, Standard Template Library (STL) support, fast dynamic casting, strong enumerations, and easy extensibility. The goal is to add useful language features while minimizing performance hits and code bloat.

Two of the features of the opC++ Standard Dialect, reflection and serialization, are quite new for C++, as these language features are difficult to construct in the C++ language. All previous attempts have been marginal at best.[1] Using opC++ is as easy as integrating the compiler, including the Standard Dialect, and writing code. Also, it works with any new or existing C++ project.

Data Reflection opC++ gives you portable data reflection, and access to metadata on a per-member basis. Simply write a visitor class with a single function and get type safe access to all data within opC++ objects. You can use these visitors to filter data members by type and other metadata.

Type Reflection opC++ gives you access to enumerated types and the full inheritance structure of your classes without requiring an instance of the type. This can be used to directly access a specific member of a type via reflection.

Serialization Automatically and seamlessly serialize your classes to and from text, binary and xml via intuitive data modifiers. All basic types and opC++ constructs can be serialized immediately, and the framework can be easily extended.

STL Support opC++ not only gives you reflection and serialization for simple data constructs, but also for all STL containers. Perform reads and writes on elements, as well as complicated manipulations with iteration, insertion, and removal access.

Fast Dynamic Casting opC++ type casting is 1000-5000% faster than C++'s standard dynamic_cast with solid time guarantees.

Strong Enumerations Use strong enumerations with automatic string conversion routines.

[1] For example, one of the only C++ serialization attempts is the C++ Boost Serialization library. This approach relies heavily on templates, and compile times increase dramatically with each new serialization path you add. Additionally, our serialization is faster than Boost's. For more details, see the opC++ benchmark white paper.

Extensible The opC++ Standard Dialect is easily extensible via extension points, an opC++ dialect language feature.

In this chapter we will introduce two new opC++ categories, `opclass` and `opstruct`, as well as several data modifiers. We also introduce `openum`, the strong enumeration included with the opC++ Standard Dialect. Although many of the data modifiers in the Standard Dialect are associated with reflection and serialization, we only introduce them syntactically here. opC++ reflection and serialization are introduced in much greater detail in Chapters 4 and 5, respectively.

3.2 Categories

The opC++ Standard Dialect introduces two new categories to C++: `opclass` and `opstruct`. Each is used for different purposes, has different costs, and different functionality. They are covered in the following sections.

3.2.1 opclass

The opC++ compiler maps `opclass` to a standard C++ class. It is a polymorphic object with fast casting and reflection/serialization support. It has private scope just like a C++ class. The syntax of an `opclass` is almost identical to a C++ class. Figure 3.1 shows an example `opclass`. You can write any code you would normally write in a C++ class (data members, operator overloads, functions, constructors/destructors, etc.) as well as some new modifiers. These will be introduced in a later section.

```
opclass foo
{
public:

    foo() : bar( false )
    {

    }

private:

    bool bar;
}
```

Figure 3.1: A simple `opclass`.

Technical Note Since `opclass` is polymorphic, be aware that it has a virtual function table. If you want a type without this cost (typically 4 bytes per object on a 32-bit system), use `opstruct`.

3.2.2 Fast Dynamic Casting

As aforementioned, opclass supports fast dynamic casting. This is achieved via the class_cast operator. Unlike C++ dynamic_cast, class_cast is achieved in constant time via a simple range check. This is possible because the opC++ compiler is aware of the inheritance hierarchy at compile time, and generates code appropriately. Figure 3.2 shows an example of using the class_cast operator. The class_cast operator will either return a pointer to the class you're trying to cast to, or null if the cast is invalid.

```
// The base class.
opclass fruit
{

}

// The subclass.
opclass apple : public fruit
{

}

...

// Here we're in code somewhere and randomly get a fruit
// object, and want to know if it's an apple.  This will
// either cast sucessfully to a fruit*, or return null.
fruit* object    = get_fruit();
apple* the_apple = class_cast< apple >( object );

if ( the_apple )
    ...
```

Figure 3.2: Using the class_cast operator.

3.2.3 opstruct

The opC++ compiler maps opstruct to a standard C++ struct. It is a non-polymorphic object with reflection/serialization support. It is well suited to any object whose type is known at compile time. If the type needs to be discovered dynamically, use opclass.

opstruct has public scope just like a C++ struct. Just as with opclass, you can write any code in an opstruct that you would normally write in a C++ struct, as well as some new modifiers. These will be introduced in a later section. Figure 3.3 shows an example opstruct.

Unlike opclass, opstruct does *not* support fast dynamic casting.

```
opstruct point
{
    float x;
    float y;
    float z;
};
```

Figure 3.3: Using a simple `opstruct`.

3.2.4 Inheritance

`opclass` and `opstruct` both require the use of single inheritance because of the way reflection and serialization are generated. Multiple inheritance is not supported. This is a common constraint in reflected languages. If your `opclass` or `opstruct` does not inherit from anything, the Standard Dialect automatically causes it to be inherited from `class_base` or `struct_base`, respectively. These are lightweight base classes that implement some necessary reflection code. All opC++ categories must inherit from one of these base classes.

For example, when one `opstruct` B inherits from another `opstruct` A that has no parent, B inherits from A which implicity inherits from `struct_base` (the opC++ compiler generates this behind the scenes). Figure 3.4 shows the example described. More details about `class_base` and `struct_base` can be found in the online reference.

```
opstruct A // implicitly is: public struct_base
{

}

opstruct B : public A
{

}
```

Figure 3.4: opC++ inheritance example.

The opC++ Standard Dialect automatically defines the typedef `Super` for every `opclass` and `opstruct`. This is a typedef for the parent class. If the `opclass` or `opstruct` in question does not inherit from anything, `Super` points to `class_base` or `struct_base`, respectively. Figure 3.5 shows an example of using `Super`.

When using `opclass` and `opstruct`, you cannot normally inherit from a C++ class or struct but instead only from an `opclass` or `opstruct`. However, there is a workaround to make this possible. Figure 3.6 shows the workaround. You can define a template class that allows you to inherit from more than one class. Inheriting from two `opclass`es or `opstruct`s is not allowed.

```
opstruct X
{
    void Init()
    {
        // ... code ...
    }
}

opstruct Y : public X
{
    void Init()
    {
        Super::Init();

        // ... code ...
    }
}
```

Figure 3.5: Using the `Super` typedef.

3.3 Modifiers

The opC++ Standard Dialect introduces several data modifiers for `opclass`/`opstruct`. Each one is presented in the following subsections. However, several are related to reflection and serialization, and are covered in more detail in Chapters 4 and 5.

3.3.1 Visibility Modifiers

In C++ classes and structs, visibility is determined by the visibility statements `public:`, `private:` and `protected:`. In opC++, you can use these keywords as modifiers on data and function statements. You can even mix C++ visibility statements with opC++ visibility modifiers. If a statement does not contain any visibilty modifiers, the visibility statement it is under is applied. However, if a visibility modifier is present, it takes precedence. Figure 3.16 shows several examples.

3.3.2 `opstatic`

In C++, static data members in a class or struct must be declared in the body of the class or struct. The programmer must always put the initialization of the static members in source code, which can be quite annoying. A C++ example is shown in Figure 3.8.

To remove this annoyance, the opC++ Standard Dialect introduces the `opstatic` modifier for data members. Using `opstatic` on a data member, you can declare/initialize the static member in one place. The opC++ compiler then generates code to the correct places. Figure 3.9 shows some examples.

3.3.3 Reflection/Serialization Modifiers

The opC++ Standard Dialect adds three data modifiers to make reflection/serialization more intuitive: `native`, `transient` and `opreflect`. Only brief examples and descriptions are given here. Reflection and serialization are discussed in depth in Chapters 4 and 5.

If a data member is tagged with the `native` modifier, it means that the data member is not reflected or serialized. This is useful for data members that are strictly internal that you want to hide. It is also useful when a type will not be known to opC++, such as a type from an API.

If a data member is tagged with the `transient` modifier, it means that the data member is not serialized, but that it is reflected and is still accessible via the reflection. If a data member does not have the `native` or `transient` modifier, it is automatically reflected and serializable. Finally, the `native` and `transient` modifiers are mutually exclusive. Figure 3.10 shows an example using the native and transient modifiers.

The third reflection modifier is `opreflect`. This modifier is a *valued* modifier - this means that it has a value associated with it. The value in this case must be a string. This modifier is used to change a data member's reflection name. Figure 3.11 shows an example.

3.4 openum

The opC++ Standard Dialect defines a new enumeration type called `openum`. `openum` is identical to a C++ enumeration with some additional features. This type has reflection support, string conversion routines, as well as strong typing. Figure 3.12 shows an example `openum` utilizing strong typing. Strong typing allows you to reduce ambiguities using the scope resolution operator.

The opC++ compiler defines three additional operators for `openum`: `count`, `min` and `max`. These are the number of enum values in the enumeration, the minimum enumeration value, and the maximum enumeration value, respectively. Figure 3.13 shows examples of using these new operators.

The `openum` stores an inner table of enumeration values and their corresponding string values. You can access a table entry using the `key_count`, `key_value` and `key_string` methods. Figure 3.14 shows an example that accesses this table. It should be noted that `key_count` is identical to the `count` operator.

There are also string and integer conversion routines for `openum`. The methods are `to_int`, `to_string`, `get_int`, `get_string`, `from_int` and `from_string`. You cannot directly initialize an enumeration with an integer as you can in C++ as this is considered unsafe. The `to_int`, `to_string`, `from_int` and `from_string` initialization functions return a boolean. The methods will return false if the initialization failed (was invalid). `get_int` and `get_string` return actual values, but you should know that what will be returned will be valid before using them. Figure 3.15 shows examples of these methods.

3.5 Alternate Prefixes

Using the keywords `opclass`, `opstruct` and `openum` can prevent some programming IDE's from performing syntax coloring and intellisense correctly. We get around this by allowing the syntax `op class`, `op struct` and `op enum` respectively. We've found that this fixes most problems, as the IDE thinks that the opC++ constructs are class, struct and enum. Figure 3.16 shows some examples.

```
// This template allows you to inherit from two different objects.
template<class A, class B>
class CppWorkaround : public A, public B
{

};

// A regular C++ class.
class X
{

};

// An opclass.
opclass Y
{

}

using opcpp::base::class_base;

// Here we inherit from a C++ class (X) and an opclass (Y).
// This works because W is inheriting from Y, which inherits
// from class_base.
opclass W : public CppWorkaround< X, Y >
{

}

// Here we inherit from a C++ class only, but
// we still must make sure to inherit from
// class_base for the opclass to work correctly.
opclass Z : public CppWorkaround< X, class_base >
{

}
```

Figure 3.6: Inheriting from C++ classes.

```
opclass Actor
{
    // This member is private (default scope).
    void Foo()
    {

    }

public:

    // This method is protected.
    protected void Bar()
    {

    }

    // This method is public.
    void DoesNothing()
    {

    }

    int           x; // This member is public.
    protected int y; // This member is protected.
    private float z; // This member is private.
}
```

Figure 3.7: Using opC++ visibility modifiers.

```
// The class is declared in the header file.
class A
{
private:

    static float x;
    static float y;
};

// The static members have to be initialized in source.
float A::x;
float A::y = 100;
```

Figure 3.8: Using the `static` modifier in C++.

```
opclass A
{
    // The declaration and initialization happen in one place!
    private opstatic float x;
    private opstatic float y = 100;
}
```

Figure 3.9: Using the `opstatic` data modifier.

```
opclass song
{

}

opclass playlist
{
    native api_window* Window;      // This data member is not reflected or serializable.
    transient int      NumSongs;    // This data member is reflected but not serializable.
    song*              CurrentSong; // This data member is both reflected and serializable.
}
```

Figure 3.10: Using the `native` and `transient` data modifiers.

```
opclass channel
{

}

opclass television
{
    // This forces the "Channels" data member's reflection name
    // to be "channel".  This also means that this is the name
    // you use when serializing it to xml, etc.
    opreflect("channel") vector< channel > Channels;
}
```

Figure 3.11: Using the `opreflect` data modifier.

```
openum Colors
{
    Red,
    Blue,
    Green,
    Yellow
}

// You can assign an openum the usual C++ way.
Colors c1 = Red;

// You can also use strong typing to rid
// the code of ambiguities.  This is not
// possible in regular C++.
Colors c2 = Colors::Red;
```

Figure 3.12: Example `openum` utilizing strong typing.

```
openum animals
{
    dog      = 2,
    cat      = dog,
    elephant = 100,
    mouse    = 0
}

// Here, the count will return 4 because there are 4
// enum values in the animals enumeration.
int count = animals::count;

// Getting the min will return animals::mouse.
animals min = animals::min;

// Getting the max will return animals::elephant.
animals max = animals::max;
```

Figure 3.13: The `count`, `min` and `max` `openum` operators.

```
openum Seasons
{
    Spring,
    Summer,
    Fall,
    Winter
}

// Here we loop through the enumeration's inner table,
// printing its contents to the standard out.
#include <iostream>

using std::cout;

int count = Seasons::key_count();

for (int i = 0; i < count; i++)
    cout << Seasons::key_value( i ) << ", " << Seasons::key_string( i ) << endl;

// The above loop will print the following to the standard out:
//
// 0, Spring
// 1, Summer
// 2, Fall
// 3, Winter
```

Figure 3.14: Accessing an `openum`'s inner table.

```
openum Cuisine
{
    Italian,
    Mexican,
    American,
    French
}

// We can convert an enum to an integer or a string using the
// to_int, to_string, get_int and get_string methods.
Cuisine c1 = Cuisine::French;

int    i;
string s;

// to_int and to_string will return false if the enum has been corrupted.
// This can happen if we did something like this, which is unsafe:
//
// c1 = (Cuisine::type) 20;

c1.to_int( i );    // i will be 3 after this function returns.
c1.to_string( s ); // s will be "French" after this function returns.

int    i2 = c1.get_int();    // This will return 3.
string s2 = c1.get_string(); // This will return "French".

// We can also initialize enumeration values from an existing
// integer or string using the from_int and from_string methods.
// These methods return a bool.  Each method will return true
// if the iniaialization was successful, or false if the int or
// string passed in is invalid (does not match an existing enum
// value).
Cuisine c2;

c2.from_int( 2 );           // This will set c2 to American.
c2.from_string( "Mexican" ) // This will set c2 to Mexican.

// Here's an invalid one..
if ( c2.from_string( "German" ) )
    // we'll never get here!
```

Figure 3.15: String and integer conversion routines for `openum`.

```
op struct X
{

}

op class Y
{

}

op enum Z
{

}
```

Figure 3.16: Alternate prefixes.

4: opC++ Reflection

4.1 Overview

In this chapter we introduce opC++ reflection. Reflection in a language means that you can get information about classes in your program at run-time. It gives you the ability to iterate over an object's data and methods, without needing to know the exact class you're dealing with.

Reflection is also often accompanied by descriptive metadata which give you additional information about data members. For example, whether or not a data member is saved to disk may be indicated by metadata.

Previously only available in non-standard C++ and other languages, opC++ adds data reflection and metadata capabilities to standard C++, opening whole new possibilities to C++ and levels of performance previously unattainable with reflection.

opC++ makes three kinds of reflection available: external reflection, instance reflection, and reflection using visitors. These are covered in the following sections.

4.2 How Does opC++ Implement Reflection?

The opC++ Standard Dialect generates its reflection infrastructure by mapping non-native data members to a parallel code structure.

4.3 External Reflection

By external reflection, we mean learning about an object (names, types, modifiers, members, etc.) without having an actual instance of the object in question.

4.4 Instance Reflection

4.5 Reflection Visitors

4.6 Costs

5: opC++ Serialization

6: opC++ Preprocessor

The opC++ compiler does not parse standard C++ preprocessor directives. It just ignores them. However, opC++ comes with a powerful new preprocessor. The default C++ preprocessor is often used for code generation and conditional compilation. However, when writing C++ macros, the existing C++ preprocessor is limiting and often very ugly. The new opC++ preprocessor introduces new syntax that allows for much cleaner/more powerful code. The following sections introduce these new language features and their syntax.

6.1 A `#include`-Like Construct: `opinclude`

When writing opC++ code, you usually put the code in a file with the `.oh` extension. However, since the opC++ compiler ignores C++ preprocessor directives, we sometimes need a way to include other `.oh` files from within an `.oh` file. To do this, we introduce the `opinclude` syntax. The syntax is shown in Figure 6.1. The filename need not have the `.oh` extension, but it is encouraged.

```
opinclude "Filename.oh"
```

Figure 6.1: Using the `opinclude` syntax.

Usually, `opinclude`-ing a file is not necessary as the opC++ compiler generates a lot of forward declarations. However, it is sometimes necessary to force the opC++ compiler to compile your opC++ code in a certain order.

6.2 A Replacement For `#define`

In C++, the `#define` keyword can be used to declare constants and macros. For example, Figure 6.2 shows a simple multi-line macro in C++. The code in Figure 6.2 looks very ugly and can be difficult to read. opC++ introduces the `opdefine` keyword to declare C++ macros more nicely. Figure 6.3 shows some example `opdefine` declarations. Note that an `opdefine` can be declared with or without arguments, just as a regular `#define`.

```
// A macro to clean up dynamically allocated memory.
#define Cleanup(p) \
   if (p)          \
      delete (p);  \
                   \
   (p) = NULL;
```

Figure 6.2: A simple multi-line C++ macro.

The opC++ compiler translates an `opdefine` declaration to a C++ `#define` statement in the generated code. It also automatically calls `#undef` on the name of the `opdefine` before generating the `#define`.

```
// A macro to clean up dynamically allocated memory.
// Note that this opdefine has arguments.
opdefine Cleanup(p)
{
   if (p)
      delete (p);

   (p) = NULL;
}

// A macro to halt execution.
// Note that this opdefine does not have any arguments.
opdefine HaltExecution
{
    assert(0);
}
```

Figure 6.3: Example `opdefine` declarations.

Using an `opdefine` over a `#define` improves readability because it looks like a function. Also, unlike regular C++ `#define`'s, you can put comments in them! For example, the code in Figure 6.4 is not allowed in regular C++. The reason this is not allowed is that C++ relies on the \ continuation character to specify the body of the macro, which when commented causes problems. Figure 6.5 shows the `opdefine` version of the macro with comments.

```
#define AddClassName(name)                \
    // This adds the name of the class.   \
    const static char* ClassName = name;
```

Figure 6.4: Comments are not allowed in regular C++ macros.

```
opdefine AddClassName(name)
{
    // This adds the name of the class.
    const static char* ClassName = name;

    /* Note: C-Style comments are allowed too! */
}
```

Figure 6.5: Comments are allowed in `opdefine` statements.

Just as `#define`'s in C++, `opdefine`'s cannot be nested. In other words, you cannot put an `opdefine` inside another `opdefine`. For coding of this kind, we introduce the `opmacro` construct in Section 6.4.

6.3 The `C++ { ... }` syntax.

When the opC++ compiler parses categories such as `opstruct` and `opclass`, it does not have information about regular C++ macros, and hence does not perform C++ macro replacement. You will almost

always use opC++'s code generation platform to automate C++ macros. However, if you want to use a C++ macro, it is possible.
In Figure 6.6, the user is trying to use an existing C++ macro to generate some code instead of using opC++'s code generation platform. This will throw an error because the opC++ compiler does not do C++ macro replacement. To the compiler, the `DECLARE_CLASS(Actor)` line of code looks like a function declaration that's missing it's return type. To resolve these ambiguities, we introduce the `c++ { ... }` syntax.

```
opclass Actor
{
    // This line of code will cause the opC++ compiler to throw an error.
    DECLARE_CLASS(Actor)

    ...
}
```

Figure 6.6: The problem of using C++ macros inside category declarations.

The `c++ { ... }` syntax passes whatever is in the braces as-is to the backend C++ compiler. In this way the macro replacement will happen correctly. It is only valid inside category declarations such as `opstruct` and `opclass`. It is also only available in certain places within these declarations. It can be either put in the modifier list of a data or function statement, or standalone as is desired in Figure 6.6. If it is to be standalone, it should be terminated with a semicolon. Figure 6.7 shows examples of each.

```
opclass Actor
{
    // Note that since this C++ macro is to be standalone, that we terminate the
    // c++ { ... } block with a semicolon.
    c++ { DECLARE_CLASS(Actor) };

    // Here we are using the c++ { ... } syntax in the modifier list of a data statement.
    private c++ { CONSTNESS } int x;

    ...
}
```

Figure 6.7: Using the `c++ { ... }` syntax correctly.

6.4 A New Preprocessor Construct: `opmacro`

In Section 6.2 we introduced the `opdefine` syntax, a replacement for `#define`, and noted that it is a bit limited. In opC++ we've introduced a powerful new macro syntax called the `opmacro`. It shares several characteristics with regular C++ macros, including:

- A concatenation operator.
- Stringize operators.

However, opmacros are much more powerful, and support the following new features:

- They are fully debuggable.
- They can generate and expand other opmacros.

6.4.1 Simple opmacros

Figure 6.8 shows the opmacro equivalents of the opdefines from Figure 6.3. Notice that the opmacros look identical to opdefines except that the opmacro keyword is used. Like opdefines, they can be written with or without arguments, and can contain both C and C++-style comments.

```
// A macro to clean up dynamically allocated memory.
// Note that this opmacro has arguments.
opmacro Cleanup(p)
{
   if (p)
      delete (p);

   (p) = NULL;
}

// A macro to halt execution.
// Note that this opmacro does not have any arguments.
opdefine HaltExecution
{
    assert(0);
}
```

Figure 6.8: Example opmacro declarations.

Using opmacros is a bit different than using an opdefine. The reason is because opmacros are expanded *before* any opC++ code is parsed. This means that you can use opmacros to generate opC++ code, or even other opmacros. They are also extensively used in most opC++ dialects.

opmacros can only be declared inside the global context (i.e., not inside of another construct such as an opclass or a class, etc.). However, opmacros can be expanded anywhere.

If you want to make use of an existing opmacro, you can expand it via the expand syntax. Figure 6.9 shows how to expand the opmacros from Figure 6.8.

```
// Expand the Cleanup macro.
expand Cleanup(ptr)

// Expand the HaltExecution macro.
expand HaltExecution
```

Figure 6.9: Expanding opmacros via the expand syntax.

6.4.2 Special Operators

As aforementioned, opmacros have concatenation and stringize operators. The concatenation operator is the @ symbol. It will concatenate two adjacent ids when the opmacro is expanded. Figure 6.10 shows an example of the concatenation operator.

```
opmacro GenerateDummyNamespace(a, b)
{
    // Generate a dummy namespace whose name
    // is the concatenation of a and b.
    namespace a@b
    {

    }
}
```

Figure 6.10: The opmacro concatenation operator.

There are two stringize operators, one for characters and one for strings. Surrounding something with a single accent character ` will transform the value to a character, and surrounding something with two accent characters `` will transform the value to a string. Figure 6.11 shows examples of both stringize operators. Do not confuse the accent character with an apostrophe. In the figure, the accent characters resemble apostrophes, but this is an artifact of the figure package.

```
opmacro GenerateSomeStrings(a, b)
{
    // This will generate 'a'.
    char x = `a`;

    // This will generate "b".
    char* y = ``b``;
}
```

Figure 6.11: The opmacro stringize operators.

Of course, the concatenation and stringize operators can also be used together, as in Figure 6.12. Concatenation is always performed first, then stringize.

```
opmacro ToConcatenatedString(a, b)
{
    // This will generate "ab".
    char* x = ``a@b``;
}
```

Figure 6.12: Using the concatenation and stringize operators together.

6.4.3 Advanced opmacros

In the beginning of this section we noted that opmacros can expand other opmacros. For example, in Figure 6.13, we use an opmacro to expand several other opmacros inside a regular C++ class. This works because opmacros can be expanded anywhere. However, if this were an opC++ category, we would just automate this using the opC++ code generation platform.

```
// This opmacro will generate a "super" typedef.
opmacro GenerateSuperTypedef(parent)
{
    typedef parent super;
}

// This opmacro will add a function that returns the name of the class.
opmacro GenerateClassName(name)
{
    string get_class_name() const
    {
        return ``name'';
    }
}

// This opmacro will expand the other two opmacros.
opmacro DECLARE_CLASS(name, parent)
{
    expand GenerateSuperTypedef(parent)
    expand GenerateClassName(name)
}

// Here we expand the DECLARE_CLASS opmacro in a regular C++ class.
class fruit : public plant
{
    expand DECLARE_CLASS(fruit, plant)
}
```

Figure 6.13: Using an opmacro to expand other opmacros.

opmacros can also generate (and expand) other opmacros. For example, in Figure 6.14, we use an opmacro to generate another opmacro and expand it. An opmacro cannot expand itself.

6.5 Other Special Syntax

This section introduces some special syntax that can be used with opdefines and opmacros.

6.5.1 Handling Stray Braces

Some programmars may wonder why the C++ preprocessor uses the \ character to format the body of macros. The reason is so you can put solitary {'s and }'s in the body of the macro. In C++, without

```
// This opmacro generates an opmacro when expanded.
opmacro GenerateMacro
{
    // This opmacro gets created when the outer opmacro is expanded.
    opmacro FirstInner
    {
        opmacro SecondInner
        {
            int* Foo;
            int* Bar;
        }
    }

    // We can expand the opmacro that gets generated.
    expand FirstInner
}

// Down here, we can expand the "GenerateMacro" opmacro.  This in
// turn generates and expands the opmacro "FirstInner".  We then
// expand the opmacro it generates, which finally generates some
// integer pointers.
expand GenerateMacro
expand SecondInner
```

Figure 6.14: Using an `opmacro` to generate other `opmacro`s.

the \ continuation character to determine the macro body, the stray {'s and }'s would make parsing brace blocks impossible, as you don't know where one brace ends and the next begins. Figure 6.15 shows an example of such C++ macros.

```
// This macro begins the Init() function.
#define InitFunctionStart \
   void Init()            \
   {

// This macro ends the Init() function.
#define InitFunctionEnd \
   }
```

Figure 6.15: C++ macros using solitary {'s and }'s in the macro body.

To be able to use stray braces in an `opdefine` or `opmacro`, we introduce the special syntaxes +{} and -{} to represent a solitary { and }, respectively. This is much more readable than the C++ equivalent. Figure 6.16 shows an example of this new syntax.

```
// Start a namespace of a given name.
opmacro BeginNamespace(name)
{
   namespace name
   +{}
}

// End a namespace.
opmacro EndNamespace
{
   -{}
}
```

Figure 6.16: Using the special +{} and -{} syntax in opdefines and opmacros.