opC++ Manual

OP Games LLC © 2007

www.opcpp.com

Contents

Chapter 1

Introduction

1.1 History

opC++ was originally created as a means to speed up computer game development. The standard programming language for games is C++ due to its speed. However, there are several nuances in C++ that can make game programming tedious. One of the most pronounced is serialization, i.e., the process of mapping class data members to memory. Writing this kind of code can also be difficult to read. We set out to write a source to source compiler transforming a new extended version of C++ (opC++) to standard C++ code that can then be compiled by an optimizing back-end compiler. In the process, we added several new features to the language that are extremely useful, but very difficult to emulate in standard C++ manually. Examples include beautification of several C++ constructs (e.g., macros), full data-mapping support for objects, state support for classes, a powerful new preprocessor that allows you to nest macros to any depth, and several new up and coming features such as interfaces/mixins/trait support. New features will continue to be added to this language as development progresses.

1.2 Overview

In this manual we describe the syntax and proper usage of the opC++ language. Since opC++ is largely a generative programming language, it should be noted that much of the syntax and usage described can be altered. What is not alterable is the validation of the code performed before the code is translated - which makes this manual very useful even when concrete examples are given. It should also be mentioned that we have the goal of making our language and compiler C++ friendly. However, we don't to the extent of making validation not possible. As a result keep in mind that the syntax is often context dependent, and what works in an opC++ class will not work in a C++ class or have the same implications when compiled with opC++. Another rule of opC++ is for the default setup to have the same overhead and performance as normal C++. You don’t need to look out for any hidden costs when using opC++, other than the ones you yourself add by hooking in generated code. The second focus of this manual is on the opC++ compiler. Topics covered are its proper usage, how you can use it to customize opC++, and how to use it to generate code. The syntax for dialects is also covered in detail in the final chapters.

Chapter 2

Constructs

2.1 opobjects

opobjects are the object oriented class construct in opC++. They may be defined with a number of keywords, the defaults being `opclass` and `opstruct`. Most custom syntax in opC++ is related to opobject contexts.

2.1.1 categories

New types of opobjects may be added by specifying user-defined opobject keywords. These opobject keywords are called categories. Categories add additional control for syntax and validation, and can help customize the code generated by the opC++ compiler. The default categories, `opclass` and `opstruct`, are in place to provide similar syntax to C++, while allowing you to write opC++ and C++ code side by side.

2.1.2 `opclass`

`opclass` is the default keyword to start a class-based opobject construct. The syntax for declaring an`opclass` is very similar to the C++ `class` construct. Figure 2.1 shows a simple `opclass` declaration.

```
// A simple opclass declaration.
opclass classname
{
    //class definition
};
```

Figure 2.1: A simple `opclass` declaration in opC++.

Inheritance for an `opclass` is similar to C++ `class` inheritance. Notable differences are that only single inheritance is currently allowed, and visibility keywords are required and not optional. Figure 2.2 makes these differences clear. The member definitions are also slightly different than in C++. This is covered in Section **??**. Finally, as in C++, by default everything is declared `private`.

```
// A valid opclass inherited from parentclass.
opclass classname : public parentclass
{
    // class definition
};

// An invalid opclass - multiple inheritance is not allowed.
opclass classname : public parentclass, public parentclassb
{
    // class definition
}

// An invalid opclass - missing visibility keyword.
opclass classname : parentclass
{
    // class definition
}
```

Figure 2.2: Examples of valid and invalid `opclass` inheritance.

2.1.3 `opstruct`

The second default opobject category is `opstruct`. `opstruct` is the keyword for opC++ structures. As an obobject category, the syntax and restrictions match those of `opclass` entirely. The only difference is that, just as in C++, its default member visibility is `public` instead of `private`. Figure 2.3 shows an example of a simple `opstruct`.

```
// A simple opstruct declaration.
opstruct classname
{
    int a;
    int b;
};
```

Figure 2.3: A simple `opstruct` declaration in opC++.

2.1.4 user-defined opobjects

Users may define their own opobject categories. These simply take the form of additional keywords. These extra keywords are syntax validated by the compiler, and placement of category keywords in invalid contexts is disallowed. These options can be enabled via the commandline or through dialects. See Section **??** for instructions on doing this. Figure 2.4 shows an example of a user-defined obobject category.

```
// A user-defined opoboject category called nativeclass.
nativeclass classname
{
    // class definition
};
```

Figure 2.4: A user-defined opobject category declaration called `nativeclass`.

2.2 `openum`

In opC++, just as we have special keywords for opobjects, we also have a custom syntax for enumerations. Parsed enumerations are called `openum`s, and have the same syntax as C++ `enum`s. Using an `openum` instead of a C++ `enum` allows opC++ to generate additional code for the enumeration. Figure 2.5 shows some example `openum` declarations.

```
// A simple Color openum declaration.
// Notice the comma after the entry "Blue" is allowed.
openum Color
{
    Red,
    Blue,
};

// A simple musical genre openum declaration.
// Notice there is no comma after the entry "Country".
openum Genre
{
    Jazz,
    Classical,
    Country
};
```

Figure 2.5: Some simple `openum` declarations in opC++.

2.3 state

One special opobject context construct in opC++ is the state. This is an alternative to writing switch statements based on the internal state of an object. The state construct is used to provide a more descriptive form for state switching code. Each state can contain function definitions. Functions with matching names and signatures will be redirected when called based on the internal state of the object. In Figure 2.6, calls to the function clicked will be routed through the internal state.

```
// An opclass definition with state support.
opclass ButtonClass
{
    state buttonup
    {
        void clicked()
        {
            // go down
        }
    }

    state buttondown
    {
        void clicked()
        {
            // stay down
        }
    }
};
```

Figure 2.6: An example of using the state construct in opC++.

By default, the internal state is tracked by a hidden data member, and may be set using the setinternalstate function. Figure 2.7 shows how to change the state of the ButtonClass defined in Figure 2.6.

```
ButtonClass b;

b.setinternalstate(ButtonClass::buttonup);
b.clicked();
```

Figure 2.7: Changing the internal state of an instance of ButtonClass defined in Figure 2.6.

Chapter 3

Modifiers

Modifiers are new keywords that can appear in front of data members and functions inside `opobjects` (see Section 2.1). Modifiers imply that code is generated for them - however by default they do not generate any code on their own. There are a number of built in modifiers which serve some special purpose - generally they imply special transformations by the compiler. Modifiers are not a free form type of meta-data like run-time attributes. Instead, they are strictly checked for correctness and only enabled by dialects.

3.1 Data Modifiers

Data modifiers are keywords on the front of a data member declaration. They imply that code should be mapped to a particular variable. The underlying meaning of modifiers is dependant upon the dialect used. Multiple modifiers may be applied to a single data member, though applying duplicate modifiers to a single variable results in an error. The meaning of modifiers may also be dependent upon the opobject category type and dialect. Figure 3.1 shows an `opclass` that uses all of the data modifiers introduced in this section.

3.1.1 `native`

The `native` data modifier implies that the data member is not mapped and not serialized. The exact underlying meaning of mapped and serialized is dependant on the dialect. Generally, the reason you want a mapped variable is for reflection, while serialized variables are a smaller subset - e.g., writing variables to a file or network stream. Though declaring a variable `native` means that it is not mapped or serialized, you may still link code to `native` variables. You may also disable the `native` modifier; this is covered in Section **??**.

```
// An opclass with data modifiers.
opclass SomeClass
{
    // Here, x is protected and native.
    protected native int x;

    // Here, y is public and transient.
    public transient int y;

    // Here, z is private, serialized and mapped.
    private int z;
};
```

Figure 3.1: An `opclass` that uses all the data modifiers introduced in this section.

3.1.2 `transient`

The `transient` data modifier implies that the data member is mapped but not serialized. You may disable the `transient` modifier; this is covered in Section **??**.

3.1.3 no modifier

When a data member is declared without a native or transient modifier, it is mapped and serialized.

3.1.4 user-defined modifiers

The real advantage of data modifiers comes from user-defined modifiers. This makes them similar to attributes in other languages, except the exact context for a modifier must be valid, and they imply behavior at compile time. The validation aspect of modifiers and modifier combinations will be explored in the future. The method to enable user defined data modifiers is covered in Section **??**. Figure 3.2 shows some possible user-defined data modifiers within an `opclass`.

3.2 Function Modifiers

Function modifiers are a concept similar to data modifiers. They are in the form of keywords on the front of member function declarations and definitions. While not so useful for serialization, they may be useful for instrumenting code or for providing reflection meta-data about functions.

```
// An opclass definition.
opclass ClassName
{
    // User-defined modifiers (reliable, unreliable,
    // server, client).
    reliable server float health;
    unreliable client string message;
};

// An opstruct definition.
opstruct StructName
{
    // User-defined modifier (compressed).
    float x;
    float y;
    compressed float z;
}
```

Figure 3.2: Some possible user-defined data modifiers.

3.2.1 `delegate`

The `delegate` modifier is a special modifier handled in opC++. It allows you to generate code for a delegate handler, while linking a default function definition to the handler. The `delegate` modifier is only valid on function definitions, and not prototypes. Figure 3.3 shows an example of this.

```
// An opclass definition.
opclass ClassName
{
    // A delegate definition.
    delegate void onclick()
    {
        // do default click stuff
    }
};
```

Figure 3.3: the delegate function modifier in opC++.

3.2.2 `inline`, `uninline`

Function inlining in opC++ is slightly different than in normal C++. In major C++ compilers, member functions declared in headers are inlined by default, or else the `inline` keyword can be used to indicate this. In opC++ you can also `uninline` functions, which removes them from the headers and places them in a single source compilation unit. In opC++, the default is to `uninline` functions, meaning major compilers will generally not `inline` the functions. The reasoning behind this is that functions should only be inlined when a profiler tells you to - this is common C++ advice to prevent code bloat. This behavior can be switched back to the C++ defaults while preserving the `uninline` keyword via a command-line option. Figure 3.4 shows an example of using the `inline` and `uninline` keywords.

```
// An opclass definition.
opclass ClassName
{
    // An inline function definition.
    inline float squared(float x)
    {
        return x * x;
    }

    // An uninlined function definition.
    uninline float squaredsquared(float x)
    {
        return squared(squared(x));
    }

    // Uninlined by default!
    void function()
    {
        // do stuff
    }
};
```

Figure 3.4: Using the `inline` and `uninline` keywords in opC++.

3.2.3 user defined

You can also create and use user-defined function modifiers in opC++. While less useful than data modifiers, they can provide instrumentation and reflection for binding other languages to function calls. Figure 3.5 shows a possible user-defined function modifier in opC++.

```
// An opclass definition.
opclass ClassName
{
    // Here a user defined modifier is used (event).
    event void incoming(string message);

    // You can mix C++ and opC++ modifiers together.
    static native void outgoing(string message);
};
```

Figure 3.5: User-defined function modifiers in opC++.

3.3 Other Modifiers

3.3.1 public, private, protected

Some of the most useful modifiers are the visibility modifiers `public`, `private` and `protected`. These can be applied to both functions and data members in opobjects. They do the same thing as the normal C++ visibility labels, but only apply to a single member at a time. They are aware of the default visibility and can be used together with normal C++ visibility labels. Figure 3.6 shows an example of this.

```
// An opclass definition.
opclass ClassName
{
private:
    public    int publicmember;
    protected int protectedmember;
              int privatemember;
};
```

Figure 3.6: Visibility modifiers `public`, `private` and `protected` in opC++.

Chapter 4

Macros

opC++ improves the way in which you can write C++ macros. It also provides a new powerful preprocessor via the `opmacro` keyword.

4.1 `opdefine`

`opdefine` is a replacement for the C++ `#define` statement. It's syntax is nicer, and it will automatically call `#undef` on the symbol to be defined. Figure 4.1 shows an example of a simple C++ macro and the corresponding equivalent in opC++. `opdefine` can also be used for macros that have arguments. Figure 4.2 shows an example of this. `opdefine` can have comments within them also, making them much easier to document than C++ `#define` statements.

```
// A simple C++ macro with undef
#undef SIMPLE
#define SIMPLE void Render();

// The opcpp equivalent.
opdefine SIMPLE
{
    void Render();
}
```

Figure 4.1: Simple C++ macro and corresponding equivalent in opC++.

One thing to notice about the `opmacro` keyword is that because its body is enclosed inside braces, you cannot use the { or } characters by themselves (without their matching brace) as this makes the code impossible to parse. This is why standard C++ macros use the line termination character \ for macros. To get around this issue, we can use `#define` to declare and use a single brace, as shown in Figure 4.3.

```
// A simple C++ macro with arguments.
#undef SAFE_DELETE
#define SAFE_DELETE(p)  \
    if (p)              \
        delete (p);     \
    (p) = NULL;

// The opcpp equivalent.
opdefine SAFE_DELETE(p)
{
    if (p)
        delete (p);
    (p) = NULL;
}
```

Figure 4.2: `opdefine` example with arguments.

```
// Declare the left brace as a regular C++ macro.
#define LB {

// The opcpp equivalent.
opdefine LOOP_START
{
    while (bKeepLooping)
    LB
        // ... add code here ...
}
```

Figure 4.3: Getting around the `opdefine` brace issue.

4.1.1 `macro`

In order to use C++ macros and `opdefine` macros in opObject code, you must prefix the macro call with the `macro` keyword. This is to help resolve ambiguities when parsing. For example, Figure 4.4 shows how to call the macros defined in Section 4.1 in opC++.

```
// Simple macro call.
macro Simple

// SAFE_DELETE macro call.
SomeObject* x = new SomeObject();
macro SAFE_DELETE(x)

// ADD_FUNCTION macro call.
macro ADD_FUNCTION
```

Figure 4.4: How to correctly call the macros defined in Section 4.1 in opC++.

4.2 opmacro

opC++ introduces a new internal preprocessor and two new keywords: `opmacro` and `expand`. `opmacro`s are declared with a syntax similar to `opdefine`s. However, unlike `opdefine`s, `opmacro`s only have an effect in the opC++ compiler. They may be defined multiple times, with each new declaration replacing the last. In addition, `opmacro`s are matched by signature (the number of arguments in an `opmacro`). `opmacro`s may only be declared in the global namespace. Finally, `opmacro`s may contain comments. Figure 4.5 shows a simple `opmacro` instantiation.

```
// An opmacro definition.
opmacro build_function(return_type,name,body)
{
    return_type name()
    {
        // Body will be replaced with the body argument passed in.
        body
    }
}
```

Figure 4.5: A simple `opmacro` instantiation.

`opmacro`s also define a small number of operators, similar to the way the C++ preprocessor works. These operators are stringize and paste. The final syntax for these operators is currently undecided, but they will probably be different than the C++ preprocessor operators for clarity.

4.2.1 expand

`expand` is an operator that allows you to expand `opmacro` definitions. `opmacro`s are only useful when referenced by `expand` calls. `expand` and `opmacro` calls may also be combined to generate opmacros from opmacros, or opdefines from opmacros. expand calls may be used in any context, and will be verified against available opmacros (by name and signature). expand calling arguments may be empty, have one or more tokens, and may include line breaks and whitespace. In addition, expand calls are more useful than preprocessor definitions in that the expanded code is fully debuggable, and will point back to the original opmacro definition via #line redirection! Figure 4.6 shows how to use expand to instantiate the build_function opmacro defined in Figure 4.5.

```
/**** Some expand calls. ****/

// This builds a void function called nothing.
expand build_function(void,nothing,/*do nothing*/)

// This builds a float returning function called process,
// which returns sqrt(20);
expand build_function(float,process,return sqrt(20);)
```

Figure 4.6: Using an expand call to instantiate the build_function opmacro defined in Figure 4.5.

Advanced uses for opmacros involve using them to build or expand other opmacros or opdefines. In such cases, special rules are used to make the expansions useful. expand calls within expand calls are only called before the outer call is expanded. expand calls within opmacros are only expanded once the outer opmacro is expanded. An example of this is shown in Figure 4.7.

```
// Note that the expand calls will not be called
// until this opmacro is expanded.
opmacro build_macros(category)
{
    expand build_element_macros(category,int)
    expand build_element_macros(category,float)
    expand build_element_macros(category,bool)
}

// The order doesn't matter here as long as the
// expansion is done after opmacros are declared.
opmacro build_element_macros(category,tag)
{
    // using the paste operator (##)
    opdefine category##_##tag
    {
    }
}

// Expand for opclass.
expand build_macros(opclass)

// Expand for opstruct.
expand build_macros(opstruct)

// These expand calls will generate the following opdefines
// (and in turn preprocessor #defines):
// opclass_int, opclass_float, opclass_bool,
// opstruct_int, opstruct_float, opstruct_bool.
```

Figure 4.7: Advanced use of expand and opmacro in opC++.

Chapter 5

Alternates

5.1 opinclude

The opinclude keyword is similar to the #include keyword in C++, except that it is used to include .oh files. opC++ ignores the #include command, but if the opinclude keyword is hit, it will load and parse the file specified. Figure 5.1 shows how to use opinclude. opC++ will look for the file to be included in every directory specified on the command line and in the current working directory.

```
opinclude "SomeFile.oh"
```

Figure 5.1: The opinclude keyword.

5.2 opdefine

See Section 4.1.

5.3 opstatic

The opstatic keyword is an opobject-specific keyword that makes it easier to initialize static class members. In standard C++, you must declare a static variable in the header file and initilize its default value in the source file. The opstatic keyword allows you to declare a static variable in the header file and initialize it in place. Figure 5.2 shows an example of this. Unlike modifiers (see Section 3), the order of keywords in an opstatic declaration matters. The opstatic keyword must be the first keyword in the declaration - this may be changed in the future. Figure 5.3 makes this clear. Finally, it is not necessary to set an opstatic variable to a value. You can simply declare the variable as opstatic, as shown in Figure 5.4.

```
/**** The static keyword in normal C++. ****/

// Header file.
class Actor
{
private:
    static bool bRenderable;
};

// Source file.
bool Actor::bRenderable = true;

/**** The opstatic replacement keyword in opC++. ****/

// Header file.
opclass Actor
{
private:
    opstatic bool bRenderable = true;
};
```

Figure 5.2: The `opstatic` keyword.

```
// Invalid syntax (opstatic is not the first keyword).
opclass A
{
private:
    public opstatic int x = 3;
};

// Valid syntax (opstatic is the first keyword).
opclass A
{
private:
    opstatic public int x = 3;
};
```

Figure 5.3: The `opstatic` keyword must come first in an `opstatic` declaration.

```
// opstatic variable without initialization.
opclass A
{
private:
    opstatic public int x;
};
```

Figure 5.4: Declaring a variable `opstatic` without initializing it.

Chapter 6

Commandline

This chapter lists all the available commandline options in the opC++ compiler. An important thing to note is that if a file or directory includes spaces, it must be in quotes! Also, separate multiple strings with commas and *no spaces*.

Example: `opcpp -oh file1.oh,"file 2.oh" -d dir1,dir2 -opobject uclass,scriptclass`

6.1 Commands

`-oh <files>`
: Input files to be compiled.

`-d <directories>`
: Input file directories, separated by a comma.

`-ohd <directories>`
: All .oh files within these directories will be parsed.

`-gd <directory>`
: Generated files output directory.

`-globmode`
: Enables opC++ glob mode.

`-verbose`
: Prints verbose output.

`-silent`
: Only prints out compiler errors and tree (if enabled).

`-fm`
: User defined function modifiers.

`-dm`
: User defined data modifiers.

`-vdm`
: User defined valued data modifiers.

`-opobject`
: User defined opC++ object types. Can only contain alphabetic charcters!

`-opobject-remove-defaults`
: Removes the default opC++ object type specifiers. (`opclass`, `opstruct`)

`-nodebug`
: Suppress debug line directives (suppress debugging redirection support).

`-opmacro-expansion-depth`
: Specifies the maximum opmacro expansion depth.

`-altclass`
: Specify alternative mappings for class categories (syntax: classprefix=category,...).

`-altstruct`
: Specify alternative mappings for struct categories (syntax: structprefix=category,...).