

## Handy Compilation Of C/C++ Methods

---

By: Paul Frazee (The Rainmaker)

### Introduction

C++ is a very expansive language with many cool features that most people don't know- that is, until now! This collection of articles is designed to teach you, the adept C/C++ programmer, the tips and tricks of C and how you too can exploit them in your daily programming. Some of the methods will range from the commonly known yet under-used macros, to the incredible expandability of inheritance, to the utter power of memory functions. Much of it will surprise you with how easy it is to implement, so read on!

### About The Author

I am Paul Frazee, amateur programmer and computer game die-hard. I am fifteen years old and wrote most of these articles in, yes, the basement of my parent's house. I have been involved any many small projects, and hope to release a commercial game some day. I use a very laid-back writing style, to make things informative but fun to read too! I hope you enjoy it!

You can email me at paulfrazee@cox.net <mailto:paulfrazee@cox.net> with your questions, comments, and corrections. Don't hesitate to contact me; I enjoy the feedback!

### Update (07/21/03):

Looking back at this article, I realize there were a lot of silly mistakes and some down-right misinformation, and all with my name on it. I apologize for not fixing them earlier, but now I have set all of that right. I am tempted to delete the above paragraph (\*sigh\*), but I will leave it there just for old-times' sake. Note also that my email has changed. Code on!

### \*\_Macros:\_\*

Macros have to be the most under-used feature for new programmers. They are very useful, and a good programmer should take note of them. Why? Well, for instance, you can streamline a function that will be called often and with congruent parameters. The windows API call MessageBox is a good example:

```
#define MSGBOX(body,title) MessageBox(NULL,body,title,MB_OK);
```

Now all you have to do is call MSGBOX("", "") instead of MessageBox(NULL, "", "", MB\_OK)! Nice, huh?

Another common use is to define a standard function prototype that can

be reproduced in an easy and clean way. For example, let's say you want to make a standard naming convention for a library you are making:

```
#define FUNC(name) void FUNC_name(void)
```

Now you can easily declare and define a function that has FUNC\_ at the front of the name, and is of type void. Observe:

```
FUNC(myfunc);                                // Declaration
...
FUNC(myfunc)                                // Definition
{
    printf("MyFunc!!\n");
}
...
void main()
{
    FUNC_myfunc();                            // Calling
}
```

The output of that program would be "MyFunc!!" Cool, huh? Now that you have seen the wonders of macros, let's get a more in-depth look at them.

### **\*\_What Are Macros?\_\***

Macros are, simply, groupings of code with a name attached. Like functions, they can have parameters passed to them, but it is important to note that they are not the same as functions. Functions are code compiled into a static memory location whose definition will appear in the actual binary output only once. Macros are, during early compilation stages, replace with the code they are defined as. Macros are pretty much always in all caps (by convention rather than necessity). You declare and define them all at once, and it is safe to define them in headers. Here is the format:

```
#define <macro name>(<parameters>) <code>
```

You do not have to surround the contents with brackets ({}), although there are certain exceptions. If you ever wish to continue a macro past its declaration line, you append that line with a back-slash (\):

```
#define MyMacro(parameter) line 1 \
                             line 2 \
                             ... \
                             line N
```

Another interesting feature is that the parameters are not type based, either. You can pass an integer, a string, or even flat out code to parameter. As long as you keep in mind what a macro truly is, you be

able to use it effectively.

In case you are having trouble grasping this whole concept, I will provide you with an example. Observe:

```
#define MyMacro(text) printf(text);
```

As you can see, MyMacro is a macro that wraps the function printf. So...

```
void main()
{
    MyMacro("Hello World!\n");
}
```

*is the exact same as saying*

```
void main()
{
    printf("Hello World!\n");
}
```

Got it? Good. And remember, macros are not magical ways around casts. In the previous example, the data passed to MyMacro would have to be a string (char array).

#### **\*\_When Should Macros Be Used?\_\***

By now you may be asking yourself why one would ever really want to use a macro, especially when a function can do exactly the same thing. Well what you should understand about the macro is that it is designed simply for code cleanliness, a tool for the programmer. In the end product, it is no different from just typing out the body of the macro. So when is best? I recommend using it in the following situations:

1. When you have a common calculation or small code snippet that isn't worthy of a whole function. I especially recommend this for situations where you want to save on speed, since a function call just adds unnecessary operations (but that is serious operation-crunching). I find macros to be especially good for calculations, since the data cast is fairly inconsequential, where in a function you would have to either change the cast or write a version for each type.
2. When you have common code that must be inserted to the calling position, such as common variable declaration.
3. When you are enforcing a syntax standard, such as in a library.
4. One word ? obfuscation.

Use well.

#### **\*\_This Pointer:\_\***

The "this" pointer is so important that I can't imagine you don't

already know of it. When programming data structures ? usually classes - it may become necessary to refer to the instance of class within itself. The answer to this problem is the "this" pointer. It is a pointer of the type of the class it is being used in, to the instance using it. Behold:

```
class CMyClass
{
...
    int int_variable;
    void MyFunction( int int_variable );
...
};

void CMyClass::MyFunction(int int_variable)
{
    this->int_variable = int_variable;
}
```

In this example, the programmer with horrible naming conventions had an ambiguous variable situation. The rules of C/C++ state that a parameter or locally defined variable overtakes a class member or global variable, which made access to CMyClass::int\_variable impossible. To specify which variable he was talking about, the programmer used the this pointer. Another common use is to pass itself into a function, like in the following example:

```
void AddToInteger(CMyClass *MyClassInstance, int value)
{
    MyClassInstance->int_variable += value;    // This Function Adds To
                                                // The Class' Variable
}
...
void CMyClass::MyFunction(int int_variable)
{
    AddToInteger(this,int_variable);    // We Pass This For The Class
parameter
}
```

Aha, very useful indeed! Note that the this pointer works just as well for structs.

### **\*\_Dynamic Memory:\_\***

As you create more complex programs, you will find that arrays of a fixed size are not cutting it anymore. Eventually you are going to need to create arrays of variable size, allocate and delete objects, and more, and I can tell you how. But first, let's have a little technical stuff. This may be over your head ? if it is, simply skip it. I would, however, recommend reading it; it is quite interesting.

First of all, you must understand that the program, when compiled, translates into memory positions in the executable (all of my discussion pertains to the PE executable file format common to Windows, though can apply in other situations). When the program is loaded into the memory, those memory positions remain intact, but are simply relative to the position the program was loaded into. So if a line of code was in position 4 in the executable file, and the file was loaded into position

1000 in the memory, that line of code would be at position 1004 in the memory.

Now, how are variables put into the memory? Well, that depends on the variable's declaration situation. If a variable is global, it is actually put into the executable the same way code is, meaning that a 4 byte integer at position 20 would take up the positions 20 through 23 in the exe memory. That is why global variables are generally frowned upon ? they make the executable take up more memory. Now, function-local, fixed-size variables (the ones you are used to) are different. They are allocated to what is called the "stack" at the beginning of a function call. The problem with those is that, as I previously mentioned, they are fixed-sized! The alternative is a dynamically allocated variable, which is managed through pointers.

Dynamically allocated variables are pointers that, instead of pointing to fixed-size variables in the memory, point to allocated memory in the "heap". Because these variables are not allocated until instructed to do so, and because the memory is managed by a flexible pointer, the programmer has the power to decide exactly how much memory the variable will need and when to allocate it. What this means for you is that you can create arrays of a size determined at runtime. I find the most common application to be with strings. Time to find out how this can be done!

### **\*\_Make Me Some Memory\_\***

There are actually a couple ways to allocate memory in C++. The most common is new and delete, the C++ commands. New allocates the specified amount of memory and returns a pointer to it, and delete deallocates it. You should never use one without the other for stability reasons. Allocate with new but forget to free up with delete? You have what is called a memory leak ? memory is being taken and not given back. Try to deallocate with delete but never allocated with a new? Your program is going to crash through an assertion, because you just tried to deallocate "bogus" or null memory. The functions' usages are as follows:

```
new <data cast>( <parameters> );    // Returns An Instance Of An Object
delete <variable>;                  // Deletes An Instance Of An Object
new <data cast>[<array element count>]; // Returns An Array
delete [] <variable>;                // Deletes An Array
```

The first two usages have to do with classes, and I will not delve into that any further. However, the latter two I will go into. You use those to create dynamic arrays ? they are very simple. Here is an example:

```
int *intarray=NULL;    // Create A Null Pointer Of Type Int
intarray = new int[50]; // Allocate An Array Of 50 Slots
...
if(intarray)          // Always Check To Make Sure It Exists To Avoid Crashes

delete [] intarray;    // Delete Intarray
intarray = NULL;      // Delete Does Not Set To Null, So It Is Best We Do
```

I recommend adhering to the example above pretty closely for allocation and deallocation, as failure to do so can very well lead to bugs.

The other way to allocate and deallocate memory is through the three functions left over from the C days: malloc, realloc, and free. How are they different? Well mostly in usage. New simplifies the allocation by calculating how much memory will be required and casting it for you (it multiplies the size of the type times the number of elements in the array). Malloc requires you to do all of that. So why use malloc and free? Personal preference, really. Here is the usage:

```
malloc( <bytes> );           // Allocate specified bytes
free( <variable> );          // Deallocate variable memory
```

Malloc, free, and realloc all make use of the void pointer (a cast-less pointer). As such, you must cast the data, like so:

```
int* pData = (int *)malloc( sizeof( int ) * 100 );
// Allocates an integer array of 100 elements
...
if( pData )
    free( pData );           // Deallocate pData
pData = NULL;                // Free does not set to null, so we should
```

Malloc may look very intimidating, but it really isn't as complex as it looks. In the end, however, I recommend using new/delete, for a few reasons. First of all, when you are instantiating data objects with constructors or destroying those with destructors (not covered in this article), the constructor or destructor will be called. The other reason is that since you are probably writing in C++, then you will want to stick to the C++ commands.

Ah, one little function left ? realloc. There will be times when you will want to resize your dynamic arrays. When this is the case, I would recommend using STL's vector or something along those lines, but I am betting that you don't know STL yet. So, we better reallocate that array! While you could just make a new array of the desired size, copy the old array's data, and delete the old array, it is much easier to just use realloc:

```
realloc( <variable>, <new_size> );
```

Realloc returns the newly allocated data as a void pointer, and also takes the variable as a void pointer, requiring some casting on your part:

```
pData = (int *)realloc( (void *)pData, sizeof( int ) * 1000 );
```

That isn't so hard! And now you too can make dynamic arrays.

One final subject I should like to touch upon. You may be asking yourself whether it is safe to mix new/delete and malloc/realloc/free. The truth is that yes, it is completely safe. Data is data, no matter

which function was used to attain it. However, it is considered to be good coding not to mix them. Now that you understand dynamic arrays, go grab yourself a book on STL and forget most of this! :D

### **\*\_Logic Without The If( ):\_\***

Standard programming languages today are built on logic. And while if statements are all good, sometimes you just need a quick expression check that doesn't require all those brackets... and so the ? was born. ? is an expression evaluator, like the if statement but designed to work as a one-liner. Here is the syntax:

```
<expression> ? <true return> : <false return>
```

That may take a little explaining. <expression> is the logic in question (e.g. `a==0` or `handle!=NULL`). <true return> is the code that will be executed if the expression evaluates to true. <false return> is the code that will be executed from the operator if the expression evaluates to false. If you are still stumped, consider this:

```
int a = 6;
int value = (a==6) ? 3 : 6;
printf("a=%d value=%d",value);
```

The output would be "a=6 value=3", because the expression `(a==6)` was true. You can also run functions instead of just returning a variable, like in this example:

```
int a = rand( )%2; // Get A Random Number (0 or 1)
a==1 ? printf("Random Returned 1!\n") : printf("Random Returned 0!\n");
```

How does that work? Well remember how macros would simply insert its code where it was used? This is kind of like a conditional macro that has its contents inserted at runtime. So if the condition evaluates to true, it is just like the true code was there, and the same for the false code when false. I find this particularly useful for when you are passing parameters or doing conditional math.

### **\*\_Getting That Log, Even In A Crash:\_\***

When you start to make a complex system, especially a game, you need to have a logging system so that you can find out exactly what made your game crash. Unfortunately, those that have made them are certain to have experienced the log-destroying crash. An undefeatable problem? I think not.

What is happening when you lose your log? Well that has to do with the way file input/output works in C/C++. When you use file writing functions, the data is not immediately written, but rather queued for writing in the memory. The actual file writing will not occur until either the file is closed or until the file is flushed. When the program crashes, the data that was queued was never written ? it was still in the memory. So as a good rule of thumb, when you write to a log, flush it immediately thereafter. If you are using the stdio FILE operations, you

do that with fflush:

```
fflush( <FILE handle> );           // Flushes output
```

That's all it takes! No more lost logs! Yay!

Whelp, that concludes the article. Hope you learned something!

Paul Frazee (The Rainmaker)