

## Article 05

### 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 [frazee@swbell.net](mailto:frazee@swbell.net) with your questions, comments, and corrections. Don't hesitate to contact me; I enjoy the feedback!

Note that this is an ever-expanding collection, and there will be updates. If you want to request a method, go ahead!

#### Macros:

Macros have to be the most under-used tool of C. They are very useful and very powerful, and a good programmer should take note of them. The most common use of them is to streamline a function that will be called often and uses similar parameters every time. The windows API call `MessageBox` is a good example:

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

As you can see, it is a much easier to call `MSGBOX("", "")` than it is to call `MessageBox(NULL, "", "", MB_OK)`.

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 fallen in love with macros, let's get a more in-depth look at them.

## What Are Macros?

Macros are groupings of code that can be run multiple times by a single call, much like functions. They are much more flexible than functions, but cannot return values. They are declared with a #define, and 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 surround the contents with brackets ({}), although there are certain exceptions (one-liners with multiple commands). Instead you end lines that are not the last lines with a back-slash (\):

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

Another thing you may find interesting is that parameters are not type based, either. You can pass an integer, a string, or even flat out code to parameter. It is how it is used that matters.

A good way to explain macros is to say that it is a grouping of code that will be inserted there instead of the macro which you call. Observe:

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

As you can see, MyMacro will print what you pass. So...

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

is the exact same as saying

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

Cool! So as you can see, macros are infinitely useful in code. Besides for being difficult to read, I don't see any disadvantages, so don't hold back!

## This Pointer:

When programming with elaborate class systems, it may become necessary to refer to the instance of class within itself. The answer to this problem is the this pointer. This is a pointer of the type of the class it is being used in, to the instance using it. Behold:

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

As you can see, in this instance it informs the compiler which int\_variable you are talking about- the one declared in the class, or the one passed into the function. How nice! Another common use is to pass itself into a function, like in this 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 C
}
```

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

## Dynamically Allocated Arrays:

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 adjust the size of an already existing array, create new arrays of a designated size on the go, etc. Fortunately for you, there are lots of functions to accommodate you, such as new, delete, realloc, and more. I will be covering a few of the most useful ones.

## The Inseparable Pair: New and Delete

Most common in dynamically allocated arrays are new and delete. New returns an array of the specified size and type, and delete destroys it. You should never use one without the other for memory and stability reasons, because trust me, you do not want to get on their bad side. Their usage is as follows:

```
new <array type>[<array size>];           // Returns An Array
delete [] <variable>;                     // Deletes An Array
```

Both of them are very simple. Here is an example:

```
int *intarray=NULL;                       // Create A Null Pointer (
intarray = new int[50];                   // Allocate An Array Of 50
...
if(intarray)                             // Always Check To Make S
    delete [] intarray;                  // Delete Intarray
intarray = NULL;                          // In Case You Reallocate
```

I displayed some of the basic rules in that example, and you would do best to follow them...

- 1) Be sure to have the brackets ([]) in delete only when you are deleting a **dynamic array**. If you try to delete a normal array or a simple pointer (in that fashion), it may/will crash the program. On the other hand, make sure that you always do have the brackets when you delete a dynamic array or you will find trouble.
- 2) Always make sure the dynamic array is not NULL. There is nothing worse than crashing your program by attempting to delete a NULL pointer / dynamic array. DirectX guys will know the delete pointer macro (if (pointer) delete pointer;), and you guys shouldn't have to be a stranger either.
- 3) After deleting a dynamic array (or even a pointer), set it to NULL. Not only is it good house-keeping, it will allow you to use that pointer later, if you see fit.

Follow these base rules and you will have a happy life of dynamic arrays.

## The Wonderful Power Of Realloc:

Realloc is a superb function that allows you to change the size of a dynamic array, while preserving its contents (as long as you don't lower the size, that is). It is superb for adjusting an array when you don't know the exact size at creation. The format is a little complex, but nothing too hard:

```
(<datatype>*)realloc(<array>,<new size>); // Returns The New Array
```

If you are still stuck, consider this example:

```
// Int_Array Will Now Have 100 Slots
int_array = (int*)realloc(int_array, sizeof(int)*100);
```

Remember to use `sizeof(<data type>)*<amount>` rather than `<amount>`, or your results will be incorrect.

## Logic Without The If():

Standard programming languages today are built on logic. Switch, if/else, and #ifdef statements rule the world of C with tyrannous might. Sometimes you just need a quick expression check that doesn't require all those brackets... and so ? was born. ?, the expression evaluator, is a deliciously powerful little tool that just doesn't get credit where credit is due. 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 value that will be returned from the operator if the expression evaluates to true. `<false return>` is the value that will be returned 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 (1
a==1 ? printf("Random Returned 1!\n") : printf("Random Returned 2!\n");
```

Very useful! And to top all, you can still set the value of a variable to what those functions return.

## Dot Dot Dot?

One of the most basic and necessary requirements of a programming language is the ability to print out variables, especially when debugging a program. It is my belief that C utilized the best way to do it with its wonderful function `printf(char*,...)`. What in the world are those ellipsis!? The answer to all your variable printing troubles, my friend.

With those handy ellipsis(...), you can create functions that work JUST LIKE PRINTF! And here's how!

```
#include <windows.h> // Or Any Header That Inc.
#include <stdarg.h>
#include <string.h>
#include <stdio.h>

void printstuff(char *text, ...)
{
```

```

        va_list arglist;                                // Ellipsis
        char buff[1024];                                // Receiving Character Bu:
        memset(buff,0,sizeof(buff));                    // Set Buff To All Nulls

        va_start(arglist,text);                          // Begin To Create Our St:
        vsprintf(buff,text,arglist);                     // Create The String
        va_end(arglist);                                // Finished
        cout << text;                                    // Display The Text (In Wl

    }
    ...
void main()
{
    printfstuff("Hey guess what! %d != 5!!\n",4);
}

```

Yes my dumb-founded friend, the output would be "Hey guess what! 4 != 5!!" Isn't that great! The uses are innumerable; you can use it for logging systems, game output, or whatever! Speaking of logging systems, here is a little trick a friend of mine taught me.

## 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 positive to have experienced the log-destroying crash. An undefeatable problem? I think not.

Let's look at the root of the problem. The log is getting destroyed, because while the file was still open and being written to, the program crashed, and the contents lost. Unless the logging system is the cause of the crash, you more than likely will not be writing to the file during the crash. The solution? Close the log when you do not need it, and then reopen it as required! To make things clean, I implement this as a class. Here is my most recent version:

```

class CLog
{
public:
    CLog(char *path);                                // Creates The System
    ~CLog();                                          // Destroys The System

    void lprintf(char *text,...);                    // Logs To The System

private:
    // Opens The Log
    void Open() { if(!logh) logh = fopen(szPath,"a"); };
    // Closes The Log
    void Close() { if(logh!=NULL) { fclose(logh); logh = NULL; } };

public:
    char *szPath;                                    // Log Filepath
    FILE *logh;                                       // Log Handle
};
...
CLog::CLog(char *path)
{
    szPath = path;                                    // Save The Path
    logh=NULL;
    logh = fopen(path,"w");                           // Open The File, Clear C
    if(!logh)                                          // If There Was An Error,
        MessageBox(NULL,"Log Error","ERROR",MB_OK);
    Close();                                          // Close The Log
}

CLog::~CLog()

```

```
{
    Close();                                     // Just In Case It Was Sor
}

void CLog::lprintf(char *text,...)
{
    Open();                                     // Open The Log File For W
    if(!logh)                                   // If There Was An Error,
        return;
    // Create The Buffer
    va_list arglist;
    char buff[1024];
    memset(buff,0,sizeof(buff));

    va_start(arglist,text);
    vsprintf(buff,text,arglist);
    va_end(arglist);
    fprintf(logh,buff);                         // Write To The File
    Close();                                    // Close The Log
}
```

Believe me, this code could save your butt more than once. Feel free to use it all you like!

Special thanks to Eugene Chabot, who taught me this and many, many other methods. Xion lives on!!

Paul Frazee (The Rainmaker)