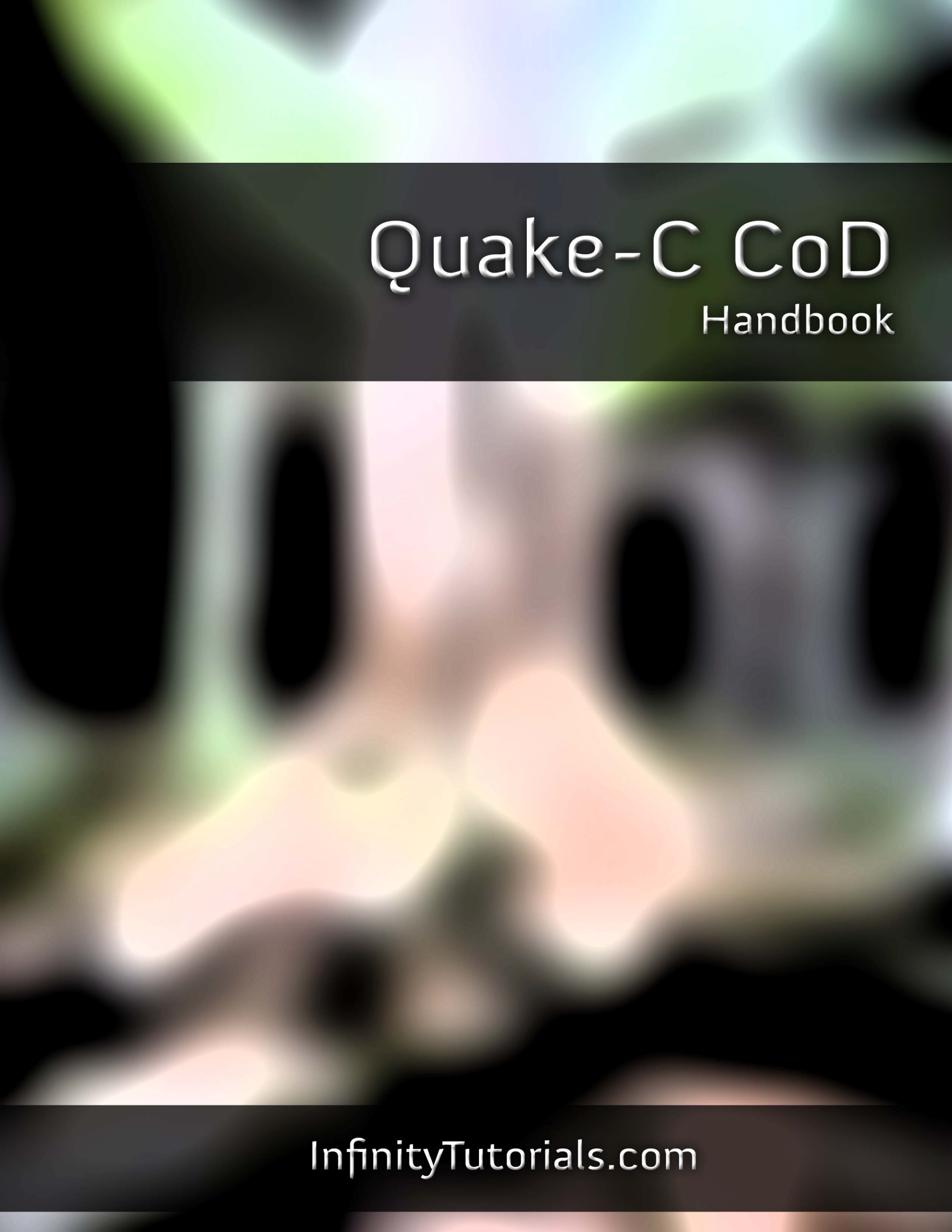
Quake-C CoD
Handbook
InfinityTutorials.com

Preface

Hi, I am Tristan. I am the creator of www.InfinityTutorials.com. I love programming, modding, gaming and best of all teaching others. That's why I strive to create a trusted modding & mapping community for everyone to join, gain experience & knowledge from.

This book should help you to understand the fundamentals of Quake-C programming for Call of Duty. By the end of it, you should feel comfortable with Quake-C, its layout, and how it works.

I hope you take away as much as you can from this book, and if you need help on any topic, or more advanced topics, please visit the website, and we'll be more than happy to help you.

So get ready, get modding and enjoy!

Chapter 1 – Variables & Operators

Welcome

Welcome to the InfinityTutorials Quake-C CoD Handbook. By reading this book from start to finish, you'll expect to be up and running making mod's for CoD in no time! Scripting isn't very hard, and as long as you're patient and you read closely, you'll be a Quake-C master in a few days with a little practice.

Alright, to start your programming adventure, you'll need a few tools. Even though these are optional, they will make your programming experience a better & more organized one. The first is a syntax highlighting text editor. I recommend **programmers notepad**, and I have included a copy for you in the **Software** folder. Using a program like this, you can create & organize projects, edit & create your scripts, and the best of all, it include syntax highlighting. If you're unsure what syntax highlighting is, here's an example:

Normal Code:

```
my_function()
{
	new_variable = "Hello World";
	self IPrintLnBold( new_variable );
}
```

Code with Syntax Highlighting:

```
my_function()
{
	new_variable = "Hello World";
	self IPrintLnBold( new_variable );
}
```

As you can see, syntax highlighting helps you to distinguish between variables, operators, strings and keywords, which we will cover later in the book.

The second tool, is a Zip compressor & extractor. Now, tools like **Win-Zip** and **RAR** will work, but what happens if you don't have them, and don't want to spend the money buying them? Well there is a great free alternative called **7-Zip.** You again can find a copy of this in the **Software** folder.

Variables

Variables are your best friend. They can hold **numbers**, **strings** or even simple things such as **True** and **False.** I don't know of a single script that doesn't contain at least 1 variable in it. This is how important they are. Variables are much simpler in Quake-C than in similar languages like C or C++. This also makes them a little less error prone, and easier to use. Here's some examples of variables and their properties:

```
my_number = 1;
```

The variable **my_number** now holds the digit value **1**. A practical use for this type of variable would be, for example, **number_of_lives** which would hold the number of lives a player has.

```
my_double = 2.0;
```

The variable **my_double** now holds the double digit value **2.0**. You could use this to store a players points for example.

```
my_message = "Hello World";
```

The variable **my_message** now holds the string **"Hello World"**. You could use this type of variable to hold a custom HUD element message, or a welcome message.

As you can see, the general format of a variable is:

```
name = value;
```

Remember to always end your lines & declarations with semi colons ";" or you will receive weird looking console error messages. In Quake-C the semi colon lets the compiler know that you're finished with one line and that you're about to start another. However, there are cases where you don't end a line with a semi colon, this is with loops, if/else statements etc.

Operators

Operators are the little things that manipulate variables. You can use operators to add, subtract, divide & multiply variables. They are extremely useful, and you're bound to use them in tons of scripts. Here are some examples of operators in use:

```
number_one = 1;
number_two = 2;
the_result = number_one + number_two;
```

In this example, **number_one** and **number_two** are added together, and the result is placed in the variable **the_result**. Thus, the variable **the_result** now holds the value **3**.

```
number_one = 1;
number_two = 2;
the_result = number_two - number_one;
```

In this example, **number_one** is subtracted from **number_two**, and the result is placed in the variable **the_result**. Thus, the variable **the_result** now holds the value **1**.

```
number_one = 1;
number_two = 2;
the_result = number_one * number_two;
```

In this example, **number_one** and **number_two** are multiplied, and the result is placed in the variable **the_result**. Thus, the variable **the_result** now holds the value **2**.

```
number_one = 4;
number_two = 2;
the_result = number_one / number_two;
```

In this example, **number_one** and **number_two** are divided, and the result is placed in the variable **the_result**. Thus, the variable **the_result** now holds the value **2**.

```
number_one = 4;
number_one++;
```

In this example, the '**++**' simply means, increase the value by **1**. You can also decrease a value by **1** using '**--**', an example: `number_one--;`

Now, what happens if you want to add **2, 3, 4** etc to a variable? Or want to multiply the variable's value by a certain number? Well, you can do it like this:

```
number = 1;
numer = number + 2;
```

In the above example, **number** is given the value **1**, then, it's set to equal itself, plus **2**. Which simply means, that in the end, **number** now equals **3**.

A shorter way of doing this is:

```
number = 1;
number += 2;
```

This shorter version above achieves the same result as the previous example. This shortcut works with all the operators. So for example, if you wanted to **divide** a variable by **2**, then **multiply** it by **5**, this is how it would be set-up:

```
number = 10;
number /= 2;
number *= 5;
```

At the end of this, **number** would now hold the value **25**.

Chapter 2 – Functions & Loops

Functions

Functions are a vital part of any programming language, and Quake-C is no different. Functions help organize your code and allow you to perform more advanced tasks.

So lets dive right in, here is an example of a small function:

```
my_function()
{
      return 5;
}
```

As you can see, functions start with a name to uniquely identify them. **Their names cannot contain anything except letters, numbers and underscores**. Their names cannot start with a number or contain spaces. They then continue with a left bracket & a right bracket. Then they continue with a left open curly brace. After this brace you type the code you want executed when your function is called. You then end the function with a right closed curly brace. One thing to note is that Quake-C isn't picky about how your code is laid out, so the following is completely acceptable:

```
my_function(){ return 5; }
```

Now, you may be asking, what does **return** do? Well, it's a Quake-C keyword, and in this example it returns the value **5.** This value is sent to what ever function called it. So, a working example would be:

```
the_number_five = my_function();
```

In this example, **the_number_five** now contains the value **5** since **my_function** *returned* the value **5** to it through the **return** keyword.

An important thing to note is that **not** all functions need to return a value, these functions are usually called **Void** functions.

Functions can perform a wide verity of operations when called. One very special feature of functions are **Parameters**. Parameters allow you to pass data into the function to work with. So lets take a look:

```
add_five( my_arguement )
{
      temp = my_argument;
      temp += 5;
      return temp;
}
```

As you can see, you can place parameters between the two brackets next to the function name. The parameter names follow the same rules of the function names. Lets see an example of this function in use:

```
not_yet_ten = 5;
this_is_ten = add_five( not_yet_ten );
```

In this example, **not_yet_ten** is initialized to **5**. Then we declare the variable **this_is_ten** and make it equal the function **add_five**, and we pass **not_yet_ten** as an argument. Looking into the function, we see it takes **not_yet_ten** and places it into the variable **my_argument**. Then, a new variable **temp** is created, and it's set to equal **my_argument**.. **5** is then added to **temp** and then the function returns **temp** which in this case is now equals to **10** since the argument passed into the function was equal to **5**. At the end of this whole procedure, the variable **this_is_ten**, equals, you guessed it, **10**!

Ok, after that long winded example, you should have a clear understanding of how parameters & arguments work. Now, another point to note is that you can have more than one parameter like so:

```
add_numbers( number_one, number_two, number_three )
{
      return number_one + number_two + number_three;
}

the_result = add_numbers( 1, 2, 3 );
```

All that's needed to use more parameters is that you separate them with comma's... simple. In that example the final variable **the_result** would equal **6**.

There's one mistake a lot of people make though, and that's using variables out side of their scope. A scope is any section between the opening brace "{" and the closing brace "}". Any variables declared inside this scope can only be used inside this scope, and not anywhere outside it. However, you can declare **global** variables which can be used out side of their scopes. Here's an example of a classic mistake:

```
function()
{
	local_var = 1;
}

result = local_var;
```

The example above would cause a compile error. This is because the variable **local_var** is declared in the functions scope, and then used outside of it's scope when assigning it to another variable. However, if you do want to perform an action like this, you can declare a **global** variable:

```
function()
{
	self.global_var = 5;
}

result = self.global_var;
```

In CoD Quake-C the above is perfect code and will not generate any errors. In that example, we used the keyword **self** which will be explained later.

Loops

Ahh, loops, my favourite little function ☺. Loops allow you to repeat a process a set amount of times, or a limited amount of times. Loops can be used to monitor variables, print messages, check players etc, anything that needs constant attention or repeating.

There are two types of loops. The first we'll look at is the **while** loop. Lets take a look at an example:

```
number = 1;
while( number < 10 )
{
	number += 1;
}
```

In this example, **number** is given the value **1**. A while loop works by checking the argument, which in this case is "**number < 10**" to see if its **true** or **false**. In this statement, it's basically saying, "*Loop While Number Is Less, But Does Not Equal 10*". If your argument is **true** then the loop performs the actions between the braces. When it gets to the last brace, it jumps back to the top, and checks your statement again. If the statement is still **true**, then it will perform the actions between the braces again. This process will continue until the statement is **false** which is this example would be when the variable **number** equals ten.

At the end of that loop, if we were to check the value of **number** we would see that it's now equal to **9**.

While loops can also be set to loop forever. This is done by simply by adding an argument that will never return false, like this:

```
while( 1 )
{
	something here…
}

OR

while( true )
{
	something here…
}
```

Since **1** is **1**, there is nothing to check, it will always be true. Even better, just put **true** in there.

Here's an example of an infinite **while** loop in use:

```
while( true )
{
      IprintLnBold( "This is a looping message" );
      wait 5;
}
```

In this example, the message "**This is a looping message**" will be displayed every **5** seconds and will never end. If you're unsure what some of those commands are, don't worry, they will be covered later.

The second loop we will cover is the **for** loop. Here is a example of one:

```
result = 0;
for( number = 0; number < 10; number++ )
{
      result += number;
}
```

As you may have noticed, the **for** loop takes 3 arguments, the first is a declaration of a variable or counter variable, the second is what to check for (same as the true/false for the while loop), the third is what to do to the counter variable when 1 loop is finished. You must separate the arguments with semi-colons.

In the example above, when the loop finishes (number = 10) the variable **result** will equal **45**. You can also make infinite **for(;;)** loops, but it's a better choice to use a **while(true)** loop in that case.

Chapter 3 – Arrays & Keywords

Arrays

Arrays are possibly the greatest type of variable alive in my eyes. Arrays are a special type of variable that can hold multiple entries of data, like a list of names, maps, scores etc.

Alright, lets look at how to make an **array:**

```
my_array = [];
my_array[0] = "Data 1";
my_array[1] = "Data 2";
```

We start arrays with the variable name, but then, instead of assigning a value to it, we assign **[]** to it (Two opening & closing square brackets). This declaration tells the compiler that the variable **my_array** is now an **array**. We then add data to the array by typing the variable name, followed by a subscript. A subscript tells the compiler which part of the array to use/assign. Array subscripts **ALWAYS** start with **zero**. This is one thing you must learn, as it's the case in many programming languages. Learn it now, and it will save you grief later on. When you add more data to the array, you must increase the subscript by **1**, and not **2** or a **fraction** etc. Here is an example of an error:

```
my_broken_array = [];
my_broken_array[1] = "Eeek";
my_broken_array[3] = "Yuck";
```

In this example error, you see that we started the array subscript with **1**, and then incremented it with **2** instead of **1**. This is bad, don't try this at home.

Now, **arrays** work great with **for** loops. Here's an example:

```
awesome_array = [];

for( int = 0; int < 10; int++ )
{
      awesome_array[int] = int + 5;
}
```

In this example, **awesome_array** is initialized, then we step into a for loop, where it sets the first **9** subscripts of **awesome_array** to the value of **int** plus **5**. So, if we had to check the value of **awesome_array[0]** it would equal **5**, **awesome_array[1]** would equal **6**, **awesome_array[2]** would equal **7** etc. This array would only go up to **awesome_array[9]**, so trying to access any subscript like **10** or **higher** would cause an error, or no error would be produced, but you'll be wondering why you're script is acting funny.

Keywords

Alright, now that you know the essential basics of Quake-C, lets move on to *some* CoD specific keywords.

1. **wait** = the time to wait in seconds
2. **self** = an alias to the entity that called the script
3. **level** = an alias to the level (the main script)
4. **waittill** = wait until another script notify's a keyword
5. **notify** = used to trigger *waittill*
6. **endon** = used to kill a function on a notify
7. **delete** = delete an entity
8. **destroy** = used to destroy structs & hud elements

These are just some of the few keywords available. Below I will give you examples of each.

wait | *this will loop every 5 seconds*

```
while( 1 )
{
	wait 5;
}
```

self | *the player who called this script will be killed after 5 seconds*

```
wait 5;
self suicide();
```

level | *grab an array of all the players*

```
players = level.players;
```

waittill | *this script will wait until "start_me" is notified*

```
self waittill( "start_me" );
self suicide();
```

notify | *this will trigger any waittill's with the same keyword, like the one above*

```
self notify( "start_me" );
```

endon | *this functions dies when "Kill_Me" is notified*

self endon("Kill_Me");

delete | *this will delete an entity*

```
model = spawn( "script_model", level.mapCenter );
model delete();
```

destroy | *this will destroy a hud elem*

```
hud_elem = newHudElem();
hud_elem destroy();
```

For a full list of all the available functions & keywords, you can go to www.InfinityTutorials.com/script or www.infinityward.com/Script/.

Chapter 4 – If/Else & Switch

If/Else Statements

If/Else statements are conditional statements that are very, very useful. They can check the state of a variable, make decisions based on data and many other cool things.

If/Else statements are actually very easy to learn. So lets dive right in:

```
my_num = 5;

if( my_num == 5 )
{
      self IprintLnBold( "The number is five!" );
}
else
{
      self IprintLnBold( "The number is NOT five!" );
}
```

In the above example, we initialize a variable called **my_num** to **5**. Then we use an IF() statement to check if the number is equal to **5**. If it is, we print the message "The number is five!", and if the number is *not* **5**, we then display the message "The number is NOT five!". If we run the script like it is, we would see the first message, however, if we changed the variable **my_num** at the top, to another number, the second message would be displayed because the IF() check would see that **my_num** does not equal **5**, and so it wall fall back to the **else** statement.

You can also do multiple levels of If/Else statements. These are If/*Else If*/Else statements. They look like:

```
num = 20;

if( num == 20 )
      self IprintLnBold( "Num equals 20" );
else if( num > 20 )
      self IprintLnBold( "Num is greater than 20" );
else
      self IprintLnBold( "Num is less than 20" );
```

Firstly, if you are wondering why the Curly Brackets have all disappeared, its because there's a neat trick with If, Else If, Else, For Loops and While Loops, and that is... if your statement will only contain 1 line of code to execute, you don't need to put curly brackets in. However, if it is more than one line of code to be executed, then you **have** to put the curly brackets in.

We see in the above example that the code will now check to see if **num** is equal, higher or lower than **20.**

Another neat trick with If, Else If, Else, For Loops and While Loops is that if you're variable is a Boolean variable, meaning it's either **true** or **false**, you can just type in the variable name to check if it's true, or put the **bang** character "!" in-front of the name to check if it's false. See below for an example.

```
is_dead = false;

if( is_dead )
      self IprintLnBold( "You are dead" );
else if( !is_dead )
      self IprintLnBold( "YAY! You're not dead" );
```

In the above example, the **if(is_dead)** is checking if **is_dead** equals true, and the **else if(!is_dead)** is checking whether **is_dead** is **false**, and in this case it is, so the player would receive the message "YAY! You're not dead".

In most programming languages, zero is considered false, and anything non-zero is true.

Some good operators for if/else statements are && (Meaning AND), || (Meaning OR), == (Meaning EQUALS) and != (Meaning DOES NOT EQUAL)

Switch Blocks

Switch blocks are usually used when you need to do multiple checks where and If/Else If statement would be too long, or messy to type out.

Lets take a look at a switch statement:

```
my_number = 5;

switch( my_number )
{
	case 5:
		self IprintLnBold( "The number is 5" );
		break;
	case 6:
		self IprintLnBold( "The number is 6" );
		break;
	case 7:
		self IprintLnBold( "The number is 7" );
		break;
	default:
		self IprintLnBold( "The number is not 5, 6, or 7" );
		break;
}
```

Now, don't get too worried about it's looks, yea, they look mean, but they can be a lot more user friendly to use that huge blocks of If/Else statements.

In the example, we start by declaring a variable **my_number**. We then start the switch block with the keyword **switch** and then in brackets what variable we will be comparing. Then, inside the curly brackets, we start the comparisons. We start one comparison with the keyword **case** followed by what we want to compare it too, then end it with a colon. Under that, we place the code we want executed if the variable equals the **case**, we then follow that with the **break;** keyword to let the script know we're done inside the switch block. At the very bottom, we add a keyword called **default** followed by a colon. Under here we put the code that will be executed if the variable doesn't match any of the cases, again, followed by a **break;**

Chapter 5 – Lets start Scripting

Let's Start

All right, you feel like making a mod yet? Well I hope you do!

To follow along with this book, you need to install the **mod tools** for Call of Duty 4. These can be found here http://www.media.iwnation.com/COD4/.

With the mod tools installed, you'll have access to all the raw files needed to start scripting a mod for the game.

If you're mainly wanting to script for CoD4, you will only need to turn your attention to the **raw → maps** folder. This is where all the raw scripts for Call of Duty 4 are located.

Whenever you see **[root]** below, its an alias for you Call of Duty 4 installation directory, which is defaults too **C:\Program Files\Activision\Call of Duty 4 - Modern Warfare.**

OK, you're ready to go!

Example Anti Camp Script

For our first task, I'll teach you how to script an anti camp function. From now on, I'll refer to the anti camp function as the ACF.

So what is needed in an ACF? Well, we need to monitor the players to make sure they are not sitting in one spot for too long. If they are sitting in one spot for too long, lets warn them that they are camping & give them 10 seconds to move. If they don't move within those 10 seconds, we'll punish them by killing them.

So already we can see we need a couple of things:

1. A while loop that constantly monitors the player
2. A function to somehow see how far a player has moved in a certain amount of time
3. A function to punish the player
4. A timer to check against.

So, lets start with our variables:

```
my_camp_time = 0;
have_i_been_warned = false;
max_distance = 80;
camp_time = 30;
```

my_camp_time will be the time in seconds the player has sat in one spot.
have_i_been_warned will either be false if they haven't been warned about camping, or true if they have been warned.
max_distance is the distance the player needs to move within the **camp_time** limit to not be considered a camper.

Now, we need to somehow check how far the player has moved on a constant basis. If they haven't moved further than the **max_distance** we need to increment the **my_camp_time** variable. Then we need to check **my_camp_time** against **camp_time**, and if they are equal, we need to warn the player & set the **have_i_been_warned** to true. We then give the player 10 seconds to move, and if he doesn't. bam! We kill him.

So, I head over to http://infinityward.com/Script/ and look for a distance function, and what do you know? They have one called **distance()** and **distance2d()**. The **distance2d()** looks like the one we need. So lets set-up our while loop:

```
level waittill( "prematch_over" );
self endon( "death" );

my_camp_time = 0;
have_i_been_warned = false;
max_distance = 80;
camp_time = 30;

while( 1 )
{
	old_position = self.origin;
	wait 1;
	new_position = self.origin;

	distance = distance2d( old_position, new_position );
	if( distance < max_distance )
		my_camp_time++;
	else
	{
		my_camp_time = 0;
		have_i_been_warned = false;
	}

	if( my_camp_time == camp_time && !have_i_been_warned )
	{
		self IprintLnBold( "Please stop camping, 10 seconds to
move" );
		have_i_been_warned = true;
	}

	if( my_camp_time == ( camp_time + 10 ) &&
have_i_been_warned )
	{
		self IprintLnBold( "You will be killed for camping!"
	);
		wait 2;
		self suicide();
	}
}
```

Now, that alone won't do anything, so, lets throw it in a custom function called **_AntiCamp()**.

Path 1:
OK, if you have the mod tools installed, go to **[root] → raw → maps → mp → gametypes**.

Path 2:
Now open a new window, and navigate to **[root] → mods**. Inside that folder, make a new folder called **MyModTest**. Inside that folder, build a folder structure like this: **maps → mp →**

gametypes. Now, take **dm.gsc** from **Path 1**, and place it in **MyModTest → maps → mp → gametypes.** Now open it in Programmers Notepad.

With it open, copy & paste your function to the bottom of the file, after all the other functions. Then, find the function **onPlayerSpawn()**, and at the bottom of the function type:

self thread _AntiCamp();

What this does is "threads" our function. Threading means the function will be called & the code will carry on without it's return happily. If we don't place the "thread" keyword there, the script would stop and wait until **_AntiCamp()** has finished, and then continue.

You can find the working mod in the **Source** folder (You can open the .zip with the 7-Zip application I included). To run a mod, start up cod4, then go to mods, select your mod and hit launch.

To test this mod out, make a new Free-For-All server, sit still for 30 seconds and watch your anti camp script in action!

Well I hope you enjoyed the Handbook, and that I have inspired you to start scripting!

Thank you for reading.

If you feel like donating, please visit www.InfinityTutorials.com and click the Donate button at the bottom of the page.