

Cam's Computer Programming Guide

Chapter 2 – C++ Programming

Version 1.0

Before you read:

I'm going to assume you have read chapter 1, or already have the knowledge on how computers and programming fit together. It is not required to know that stuff, but it is recommended.

In this chapter:

In the C++ Programming chapter of my manual/book/tutorial, you will learn about the C/C++ programming language. I am going to refer it as "C++," because as you know (because you've read the intro, right?), C++ is the successor of C, so we will be writing C++ code. By learning C++ you will also know C, only some code won't work in a C program, because it is C++ only code! Besides that point, after reading through this chapter, you will have a really good amount of knowledge on how to program in C++. This chapter is REALLY IMPORTANT! Skipping this will ultimately kill the point of my guide. You NEED to know this stuff in order to do everything in the rest of the chapters. If you feel you know C++ already, skip this. Otherwise, read it. Even if you feel you know C++ it's always a good idea to make sure you know everything, or even refresh your memory.

Why use C/C++ programming language?

As I've said in the previous chapter, it is currently the industry standard. For beginners, it might be a little too complicated and confusing. Because of this, I'm going to try and make it as simple as possible. I'll also give you many working code examples. The programs you will create in this chapter will be cross-platform without using SDL, which means the programs will be simple. In the next chapter, you will learn how to make a simple (a non-console based) application using SDL. Windows programming will NOT be covered in my tutorials, since my main focus is programming cross-platform applications; mainly games. Next, you will make a simple game. After that, we will start getting fancy using OpenGL and make a 3D game. If you don't want to make a game, you can stop at chapter 3; since you won't need any further knowledge to make programs, that are not games anyway. Other than that, you should catch on to programming using C++ quickly, and it will become very... very easy!

Now for the fancy reasons! You can use C to program just about ANYTHING! Things from washing machines to any microcontroller. You can write scientific systems, even operating systems. C/C++ gives you the maximum amount of control over anything. There are unlimited possibilities! It can change your life :)

Understanding how to code:

Programming doesn't magically work. You can't just type random code everywhere, it has to be organized and structured!

Semantics: Human-readable "words" that can be easily converted to machine code.

Syntax: The "grammar" you must follow in order for it to be understandable

Every “chunk” or “block” of C++ code is contained in a set of braces. Yes, braces. No, they don't go on your teeth! There is an “opening” brace, and a “closing” brace. Think of it as a container that closes off your code, or your spaghetti! So, it should look like this: { }. One opening brace, one closing brace. Inside those braces you will place your C++ code. Now, how do you get access to this code? You have to have some sort of *trigger* or *command* that “opens” the “lid” to your “spaghetti.” This, and every other line of code, is called a *statement*.

Let's look at this at a different angle. Let's say you have to label your container spaghetti, so you know there is spaghetti inside it. This “label” is also what opens your “container.” Here's an example:

```
spaghetti()  
{  
    // spaghetti goes here  
}
```

Before you freak out and decide to run away, let's focus on my point before we go into what everything is. So the label to our spaghetti container is “spaghetti().” You can use this label to *execute* the *script* or *code* inside the container. You do this by typing the container name and then adding a semicolon to the end like this:

```
spaghetti();
```

That will run whatever is inside the *function*. WOAH! FUNCTION??? Yeah, I'll go over that in a bit. What's with the parenthesis? I'll go over that too.

Comments:

I bet you are wondering that the double slashes mean, right? Well, they create a *COMMENT*. Comments are very useful, use them A LOT! Whatever follows the double slashes will not be compiled, so you can put anything you want on the rest of that line. You may be coding, then give it a rest for a week, come back, and say to yourself, “Huh? What does this do???” Which is why you should use comments! Also, they are good for debugging. You can cancel out a line of code just by putting double slashes before it!

Multiple line comments are also useful. If you are programming in just C, you cannot use double slash comments, so you are stuck using only *multiple line comments*. These are created by using */** and **/*. */** starts the comment, and **/* ends it. This is handy if you want to cancel out a huge chunk of code and don't feel like putting */*** before every line!

C++ Syntax:

C++'s “grammar” makes it so you have to *declare* the *function* before you can *execute* it. Think of it as this: you have to know what's in your drink before you drink it. It may be poison, or worse, your least favorite drink! So this means before you execute the code, place the *definition* (the one with the braces) before you run the code.

Now, in C++, functions don't run in the order you've placed them, unless they are in the *main loop*. The main loop is what is actually executed, and it is run before anything else is. The main loop is required in every program you create.

The Simplest C program, ever:

Here's the simplest C program (that does something fun) you can have:

```
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

Alright, now let me explain. Let's start from the top. The *#include statement* "includes" a *header file*. Header files are very useful, and they are used to add functions to a program. Inside `stdio.h`, there is a declaration for the `printf()` function! Otherwise, there would be a compilation error, because `printf()` would not be defined! Next line, this is our main loop! What does `int` mean you say? Simply put, it means integer. Integer is a *type define*. This means the main function *returns* and integer. That, I will explain in just a moment.

NOTE: `stdio.h` stands for "standard input/output" header. It is used for input/output functions.

Now for the "function." A function is just what it says. Symbolically speaking, a function could be "flip light switch on." Another would be "flip light switch off." It is a block of code that executes and does something. In this example we are using `printf()`, which prints text to the screen.

Inside the parenthesis is where you place function *parameters* or *arguments*. Symbolically, a parameter is an adjective, or a *modifier*. For example, a red balloon, or a brown balloon. Red and brown are the balloon's parameters. This can also be how hot something is, from a scale of 1-10. For `printf()`, the parameters are: a *string*, and *variables*. A string is a line of text, enclosed in quotes. A variable can be an integer, a *character*, anything. When I say "character," I mean a single letter, such as A or B. More on parameters later in the functions section.

NOTE: You can remember it better by using the word "arguments," because it "argues" with the functions!

The `\n` inside the string is a *string modifier*. There are many. What this does is create a new line. Here is a list of some string modifiers:

- `\n` – creates a new line
- `\t` – creates a tab
- `\"` – creates a double quotation mark
- `\'` – creates a single quotation mark
- `\\` – creates a backslash
- `\0` – null
- `\b` – backspace
- `\?` – question mark

You're probably wondering why we need these can't just type " or \ right? It's because they are used for modifying the string. Typing " will simply close the string.

Something noteworthy: you MUST type a semicolon after EVERY variable declaration of function call! It says to the compiler "I'm done with this, next!" You see, when you compile your code, white space cancels out. So the simplest program would compile like this:

```
main(){printf("Hello world!\n");return 0;}
```

I left out `int` and `#include` on purpose. `#include` is read BEFORE the compilation process. It is a pre-compilation code. `int` just tells the compiler what kind of a function it is, and it generates machine code based on that.

Something of obsession: in C++, white space is NOT ignored for case. Everything in C++ is case-sensitive. Here's an example:

```
int WristWatch;
```

is not the same as:

```
int wristwatch;
```

In a nutshell, you can't be too careful with white space!

So, what is it with these "type declarations" anyway? Now it's time to know. On the last line, there is a return statement. This "returns" something back to the function it is inside. It is inside the main function, therefore it returns 0 to the main loop. 0 is an integer, so in order for it to get the 0 back, the function's type must be `int`. As I've said in the intro, 1 means on and 0 means off. So if something is ON it must be TRUE, and if something is OFF it is FALSE. This is how *boolean operations* work, which I will go over later. I made this note because any *loop* that gets the return value of 0 (aka false), it stops running. I will go over loops later. For now, just know, once a loop or function reaches the return value, it stops running. This is why the program exits when you return anything! In this case, the 0 in the main loop means NO ERROR. Return values can be used for error checking as well! Return values are very useful, which I will go over in detail later.

Variables:

Correct, it is time to go over variables in detail! Yay!!! First, let's find out what variable means. A *variable* is like a *storage tank*. They let you store values into the memory! Okay, lets see... now I will let you know all of the types of variables, and what they do. We will examine this using a chart:

Variable:	Example:	What does it do?
<code>int</code>	1	Simple counting number, can be positive or negative.
<code>unsigned int</code>	1U	Counting number. Can only be positive.
<code>long</code>	10L	Never used anymore, used to be a larger version of <code>int</code> .
<code>unsigned long</code>	10UL	Non-negative long.
<code>float</code>	1.0f	Precision real number. Smaller memory use than <i>double</i> , but not as accurate.
<code>double</code>	1.0	Standard floating point real number. (aka decimal)
<code>char</code>	'c'	Single character. Placed in single quotes or no quotes.
<code>string</code>	"string"	Line of text.*
<code>bool</code>	true	Logical boolean operating. Can be either true or false. Equivalent to 1 or 0.

covered later.

Let us now learn how to declare variables. Here is an example:

```
int integer;          // standard declaration
char a, b, c;        // you may declare multiple variables on one line
bool frog;
```

Simple. Just add the variable type, the variable name, and then the semicolon. You CANNOT have two of the same variable names! So this would create an error:

```
int frog;
char frog;
```

Even though they are different types, one cancels out the other, and it results in an error. Now how about giving things some values? It's simple. Just use the equal sign!

```
frog = 7;
```

That sets frog equal to 7! You may also set a value upon declaration as well:

```
bool red = false;
```

Yes, it really is that easy! Now... you need to know about *globals* and *locals*. Globals are variables declared outside the main loop and outside any functions. Locals are variables declared inside a function or the main loop. Here's an example:

```
int global;          // the global

int eat()            // random function
{
    int local1;      // declare a local
    local1 = 7;      // can do this
    local2 = 6;      // cannot do this!
    global = 8;      // can access global from anywhere!
}

int main()           // main loop
{
    int local2 = 6;   // declare a local
    eat();            // run the eat function
    global = 7;       // reset the global to 7
}
```

In the above program, "global" is the global, because it is outside the code blocks. Both "local1" and "local2" are locals, and CANNOT be accessed outside of the blocks they are in. So in eat function, local2 = 6; would result in an error!

File types and their uses:

To understand C++, you must know the file types that work with the language. There are two main types: "*.cpp" and "*.h". The .cpp files stand for "C plus plus" files. These are the files that will be compiled and executed. The second type, are *header files*. They include *function prototypes*, which will be discussed later. Header files can be very useful, as you will see later. You would place code such as the "simplest c program, ever" as said above. The stdio.h file is a header file, as it

includes prototypes of functions, such as printf(); Note, if you don't include the header file, you cannot use the printf(); function, as it will tell you it is undefined! The only other file type you would see while exploring programming, is the "*.c" file. This is just a standard C source file. This is the same as the .cpp files, only you cannot use C++ functions! We will be using .cpp files instead.

Binary arithmetic:

Here is where your binary number knowledge comes in! Let's look at all of the different *operators*. Operators are the same as they are in math: addition, subtraction, multiplication, etc. Let's look at all the different ones in a beautiful chart:

Operator: (name)	Example:	Function:
+ (add)	1+1	Adds variables together.
- (subtract)	1-1	Subtracts variables.
* (multiply)	1*1	Multiplies variables.
/ (divide)	1/1	Divides variables.
% (modulo)	1%1	First grade division. Value of the remainder ONLY.
++ (increment)	1++ or ++1	Increases variable by 1.
-- (decrement)	1-- or --1	Decreases variable by 1.
= (assign)	frog = 7;	Used to assign variables values.
+= (special add)	frog += 2;	Adds number on right to number on left.
-= (special subtract)	frog -= 2;	Subtracts number on right to number on left.
*= (special multiply)	frog *= 2;	Multiplies number on left by number on right.
/= (special divide)	frog /= 2;	Divides number on left by number on right.
%= (special modulo)	frog %= 2;	Modulo divides number on left by number on right.

How do you use them? Here you go:

```
frog = 7 + 8;      // sets frog equal to 7 + 8 (which is 15)
frog += 15;      // does the same as the above, only its shorter
frog++;         // adds 1 to frog variable
frog = 7%6;     // frog = remainder of 7 divided by 6
frog %= 6;      // frog = remainder of frog divided by 6
```

Pretty easy? I sure hope so. I think they are pretty strait foreword. Don't forget you can use parenthesis in your mathematical equations:

```
frog = (7/6) + (8+9-(12%7)*8);
```

Anything you can do in math, you can do with binary numbers in C++.

Logical operations:

Programs would be boring and really simple if we could not compare things logically! Logically means in your head. Not mathematic. You could not answer the question, "Are you a programmer?" in numbers, can you? I didn't think so. We will now discuss some operators that will help us out in this situation! Here is another beautiful chart to help you out: (stare in awe!)

Operator: (name)	Example:	Meaning:
== (is equal to)	if(1==1){var=true;}	Equality. A check to make sure the left has the same

&& (AND)	if(1&&2==frog){v=1;}	Same as AND in chapter 1.
(OR)	if(1 2==frog){v=1;}	Same as OR in chapter 1.
! (NOT)	frog = (!frog);	Same as NOT in chapter 1.

These should be pretty easy. You are just about doing the same thing as you were above with the regular operations, only you are COMPAIRING things LOGICALLY! The only thing you most likely wouldn't understand (unless you were psychic or a genius) is the "if" going on in the examples!

Loops:

Loops are ways of using logical operations to greatly enhance your programs. There are many, so let's go over them:

```
if(condition)
{
    // do stuff
}
```

Just like algebra, this loop basically says, "if condition is true, then execute code in braces!" In algebra, you use "if then else" statements. You do the same in C++:

```
if(!condition)
{
    // do stuff
}
else
{
    // if condition is true
}
```

If all "else" fails, execute the code in the else loop! You can do the same for all of the loops below as well! Now for the for loop:

```
for(initialization; condition; increment) // 3 possible arguments
{
    // do stuff
}
```

This one is special, it executes code as the fore loop is started, then checks for condition, and then increments. You can leave any or all of them blank if you need.

```
while(condition)
{
    // do stuff
}
```

This one continuously executes its code and never leaves the loop (unlike if, which runs the loop once, then exits) until the condition returns false, or the loop is broken by command.

```
do
{
    // this code executes AT LEAST ONCE
}while(condition);
```

Code is executed once, and loops back if condition is true.

Switches:

Switches are very useful, they check if something is true, and execute code based on that. Think of it as an on-off switch for turning off the lights:

```
switch(light)
{
    case ON:
        color = 6;
        break;

    case OFF:
        color = 3;
        break;

    default:
        // if nothing is detected (light neither on or off)
        break;
}
```

This executes with the variable in the argument. It then checks if the “case” is something. If it is, then it executes code. If it’s none of the cases, it executes the default code.

Functions:

With your newly gained knowledge, I’d like to go over functions in detail. Firstly, we need to go over arguments (I prefer this term over the term “parameters”). You can easily link local variables from inside other functions using them. Normally, it would be impossible! Using arguments is VERY useful! I can’t tell you how many times arguments have saved my life. To see what I mean, look at this example:

```
int grab_local(int grabbed_local)
{
    grabbed_local++;
    return grabbed_local;
}

int main()
{
    int local = 7;
    int new_local;
    new_local = grab_local(local);
}
```

This may be a little complicated, so I’ll explain it the best I can. When you create an argument, it must have a type. Once the argument is created, you can use it inside the function. You can “thread” a variable into the argument, and use it inside the function. So in this example, I am threading the local variable (which you can’t normally touch outside the block of code it is in) through the outside function. This way, I can alter the threaded variable and then return the altered version. I am assigning the returned altered local variable to the new_local variable. This can be very useful; you will see what I mean once you start programming big things such as games!

Now for something of good practice, yet a little more advanced. It is now time to discuss... *function prototypes*. You can pretty much guess what this means. But I’m

going to explain how these are useful. Using prototypes, you can access any function, anywhere in your program! The only catch is you need to use *header files*, and you must declare the function in the header file. You can then define it in any C++ file you choose. For example:

```
#include "header.h"
#include <stdio.h>

int main()
{
    function();
return 0;
}

function()
{
    printf("Hi.");
}
```

Apparently, the function() function was defined AFTER it was used. Luckily, we have included the header.h header file, which contains the prototype. Here is what the header.h would look like:

```
function();
```

Yep. That easy! It's just the first line of the function with a semicolon at the end! If you want to create a function that returns nothing, define it using void:

```
void this_returns_nothing();
```

NOTE: for #include files, use < and > for a *system include file* and quotes for a header file in the *source directory*! More on directory structure later...

Overloading functions:

In C++, you can declare a function more than once, as long as it has a different type. This is called *overloading*. Here's an example:

```
int dupe(int x);
float dupe(float x);
bool dupe(bool x);
```

In this case, dupe stands for "duplicate". It would be a lot easier to do this:

```
int x = 7;
float y = 8.0f;
bool z = false;

dupe(x);
dupe(y);
dupe(z);
```

Than to create separate functions do handle each input type, like this:

```
int dupe_int(int x);
float dupe_float(float x);
bool dupe_bool(bool x);
```

When you could just easily overload the function.

Inline functions:

Normally, programmers avoid function calls, as they slow down the program. It is advised to use functions for a large chunk of code ONLY! Inline functions were made for small 1 or 2 line function calls. Instead of running the function itself, the compiler "copies and pastes" the code in it's place, and runs like a function. This is a major improvement in speed. Declare inline functions as you would regular functions, only place "inline" before anything else:

```
inline int fast_function(int x)
{
    printf("X is: %d", &x);          // do something random...
}
```

Arrays:

Here we go. Arrays are structures that handle variables into a linear "array" of elements. This makes life 10x better! Instead of doing this:

```
int a_0, a_1, a_2, a_3, a_4;
```

You can simply do this:

```
int a[5];
```

It's the same! Please note that there are 5 "layers" that start counting from zero. The real name for it is *index*, but that's confusing; so let's just call them layers! So the layers are 0-4, which counts as 5. You can store a separate variable on each layer, depending on the type:

```
a[0] = 1;
a[1] = 2;
```

Now, think about it... if you had an array of multiple characters, it would make a string, wouldn't it? This is the same as a string, except you don't have to include the string header!

```
char string[30] = "Hello people!";
```

The string can have up to 30 characters maximum, unless you increase the storage space. If you want to get fancy with math, you can create matrices! Oh the joy! Here's an example:

```
int matrix[3][2];
```

This is a matrix with "2 rows and 3 columns." If you are into math a lot, then this will come in handy! Enjoy, my friend. You can do this as much as you want, of course. So this means you can have three, four, or whatever dimensions.

Pointers:

When something is stored into the memory, it is placed in a certain address. Sometimes, we need to know that address. We find that out using *references* and *pointers*! Yay! Pointers can be annoying or really fun depending on the way you treat

them, and whether you understand them correctly or not. A pointer is a variable that points to another variable's address. In other words, a pointer points to another variable. So if the variable a pointer is pointing to changes, so does the pointer's value. This can be very useful. Here's an example of a pointer declaration:

```
int *p, i;
```

A pointer can have any type, and it is declared using the asterisk. Note the `i` next to the `p` has no asterisk. This simply means, it's not a pointer!

A reference returns a pointer's address. This is exactly how you can make a pointer point to another variable. Let's make `p` point to `i`:

```
p=&i;
```

A reference is recognized by the ampersand character. If you forget to do so, and do this:

```
p=i;
```

You are making the pointer equal the value of `i`. This is an error, because a pointer can only hold an address. Other than that, if you do it correctly, you can then assign the pointer a value. It won't change the pointer's value, but rather, it changes the value or the variable it points to!

You may use pointers to create strings, as they POINT to a line of text, and don't store it! Therefore, an error-free, really easy way of doing strings:

```
char *string;
```

As you may notice in a lot of code I create, I use this method the most. It's easy, fast, and doesn't waste precious memory!

You may not think much of pointers right now, but later along the lines of programming you will see how useful they can be!

As a side note... there can easily be errors. You do this by declaring a pointer, and assigning it a value. This gives the value to a random memory location, since the pointer isn't pointing to anything! This will result in an explosion.

Structures:

This section is advanced, so it may be a little bit harder than the rest. Structures are like "objects." This is where object-oriented (or OO) programming came in (which I will go over in more detail in another tutorial). This is the same as a class, which I will go over next. I don't use classes, they are just more complicated to do just about the same thing as structures. Let's say we are making a game. We want to create a ball, but we must have a ball "object." So now let's design the ball's STRUCTURE.

```
struct ball          // our ball structure or "object"
{
    string name;     // the ball's name
    int x, y;        // the ball's x and y location
    int color;       // the ball's color
}
```

```
    int health;           // the ball's health
};                        // DON'T FORGET THE SEMICOLON AT THE END!
```

Then, you would create the *object*.

```
struct ball green_ball; // instance of ball object
```

Then, you can access variables, functions, or whatever is in your structure by typing the object name, then a dot, and then the variable or function:

```
green_ball.name = "Johnny";
green_ball.x += 1;
green_ball.y = green_ball.x + 7;
green_ball.color = green;
if(damage==true)
{
    green_ball.health -= 5;
}
```

You can do anything you want, even create more objects! You can also make a function structure with arguments! Here's an example:

```
struct ai_ball explod(int x, int y)
{
    draw_blood(x, y);
    damage(x, y);
};

struct ai_ball new_ball;
new_ball.explod(x, y);
```

You can also have pointers to structures:

```
struct p_ball
{
    int x, y;
};

struct p_ball *pts;
```

Pointers can be annoying looking in structures:

```
(*pts).x = 7;
```

Since pointer structures are very common, useful, and unfortunately ugly, the creators of C decided to make a much prettier way:

```
pts->x = 7;
```

Much better! It's also absolutely 10x better and easier to understand, since it "points" at its value. Plus it just plain looks sweet! Here's an interesting fact: before C++ came out, the -> notation was rarely used, so it confused many experienced C programmers!

Okay, now how are you going to define the functions inside the structure? Easily! Just type the structure name, followed by "::" which is followed by the function

name. Here's an example using the structure above:

```
ai_ball::draw_blood(x, y)
{
    // whatever the function does
}

ai_ball::damage(x, y)
{
    // whatever the function does
}
```

It really is that simple... you can also create *constructor* and *destructor* functions. A constructor is executed when the object is created, and the destructor is executed when the object is destroyed. You define them by using the structure's name as a function, without a type or return value. Here's an example:

```
struct sample_structure
{
    sample_structure();           // constructor
    ~sample_structure();         // destructor
};

sample_structure::sample_structure()
{
    // code executes upon object creation
}

sample_structure::~~sample_structure()
{
    // code executes upon object destruction
}
```

As you can see, this concept is very simple, easy, and effective. More on OOP (object-oriented-programming) later!

There are much more tricks to structures, but I will go over those once they pop up as I need them.

Classes:

You may see the word class followed by confusing words such as "public" and "private." Honestly, don't worry about classes. Class is the same as a structure, only it's more organized. In the public section, there are functions and variables available to the "public," or in other words, to whatever uses the class. The private section contains functions and variables that are ONLY usable to functions inside the class (up in the public section). Once again, I don't use them. No clue why, but I just like the word struct better, plus I don't have to type that extra "public" line!

Enumerated constants:

These lovely bits enable you to create new types, and define variables for that type. These variables have a set value. Let's say you wanted to create a type that defined the color of something:

```
enum COLOR { RED, GREEN, BLUE, BLACK, WHITE }
```

This creates the COLOR type, and defines RED the value of 0, GREEN the value of 1,

BLUE the value of 2, and so on. If you want to give them all a specific value, then do so:

```
enum COLOR { RED=50, GREEN, BLUE=70, BLACK=80, WHITE=90 };
```

This gives RED 50, GREEN 51, BLUE 70, BLACK 80, and WHITE 90. When you want to use them, create an "object" (like a structure) of the enumeration:

```
COLOR frog_color;
```

Then use that object by using an argument:

```
int frog_c;           // the frog's color
frog_c = frog_color(50) // set the frog's color to red
if(frog_c==RED)
{
    printf("The frog is colored red");
}
else
{
    printf("The frog is not red");
}
```

See? Easy as cheese! Enjoy.

Directory Structure:

Every good program has an easy to use, fast, and efficient directory structure. This is the way I do it, and the way I recommend you do it:

Project Directory:

```
ProgramNameFolder
  bin
  dev
  win32
    vcpp
    devcpp
  macosx
  linux
  ..
docs
include
lib
  win32
    mingw32
    vcpp
  macosx
  linux
  ..
media
obj
src
test
```

Inside your project directory (preferably C:\Projects), create a folder with the name

if your program. Inside that folder, make all of the folders listed above. Here's a description of what each one's purpose is:

bin: your finalized, ready to distribute files (executables, .dll's, media, etc.)
dev: inside this folder will be the project files for your compiler program, for each platform
docs: place all documents inside this directory
include: all include files needed for external libraries (such as SDL and SDL_image)
lib: all library files (.a, .lib, etc.) needed for .dll files & etc., for each platform
media: all images, sound effects, etc. go in here
obj: compiled .o files from the compiler (set up compiler to place them here)
src: all .c, .cpp, .h, and other source code files
test: testing/debug directory

This directory structure, in theory, is bulletproof. I have used it multiple times, and it is the best.

Libraries:

Think of libraries as a huge pre-compiled "stash" of functions that you can grab any time you want. Libraries are contained in .dll files, which stands for Dynamic Link Library. Plug-ins are good examples of wise library usage! Libraries make your programs more flexible.

Although libraries are very lovely, they are a bit complicated. Let me explain them as easiest as possible. To create the library, make a project with your compiler that compiles the code into a .dll file.

For libraries to work, you **MUST** have header files for every source file! Unless, of course, you create one big include file with every function prototype, which works just the same. You will need these later, as I will explain in a bit.

Here's a sample .dll program:

```
// util.h - utility function prototypes

extern void useless_function()
extern int sample_function(int x, int y)
```

The above is the source for the util.h header file. Take note of the *extern* placed before the functions. This lets the compiler know it is an external function that will be used later. When I say later, I mean inside the program using the compiled .dll file. Now to define the functions:

```
// util.cpp - utility functions definitions

#include "util.h" // include the function prototypes

void useless_function()
{
    // do whatever
}

int sample_function(int x, int y)
{
    int z;
```

```

    z = x+y;
    return z;
}

```

This is your .cpp source file for the library. You will then compile both files into a .dll file. The compiler should output static link library files (.a, .lib, etc.) which the compiler needs when compiling the program that uses the .dll file.

You should have read the above section about the directory structure. If not, go ahead and do that right now.

Once it is compiled, place the .dll in your test directory (where the .exe will be placed), and place the .a/.lib files into your lib directory. Finally, place all include files (in our case, just util.h) into the include directory. Now you are ready to compile your program that uses the .dll!

Setup your compiler linker to link the .a/.lib files created from compiling the .dll. If you don't know how to do this, check your compiler's documentation. After that, make sure you set up the compiler so it uses the include directory to include the files in the folder when requested.

Now you should be good to go. Here's a sample program that could use the .dll file you have compiled:

```

// main.cpp - main program loop & init

#include <stdio.h>          // include the standard input/output header
#include <util.h>          /* use < and > because it should be a system
                           variable, if you set it to include our
                           include directory! */

int main()
{
    int a = 7;
    int b = 4;
    int c;
    c = sample_function(a, b);
    useless_function();

    printf("C is: %d", &c);

    return 0;
}

```

Hopefully, this should compile and work if you did everything correctly! That's all there is to it! These can be very useful. A good example of a very useful library is the SDL library, which we will be using a lot in the near future!

Other random things you may need to know:

typedef:

Normally, you only have the C/C++ types, such as: integers, characters, etc. How do you create your own types? Easy, using typedef! Here's an example:

```
typedef unsigned short int USHORT;
```

This creates a type named "USHORT." This is so you don't have to keep typing

“unsigned short int” before any new variable. You can also use typedef for many other things.

#define:

You may see this a lot, since it is very useful. You can use #define to create an unchangeable global variable. Unchangeable as in, once it’s “defined,” you cannot re-define it. When I say global, I mean global; global as in you can use it for ANY source file! This is useful for creating macros.

#ifdef, #ifndef, & #endif:

These are exactly what they “say.” #ifdef means “if defined.” This can be very useful, especially for cross-platform applications. Let’s say you want to include the windows header file if the user is using a Windows platform. You would put:

```
#ifdef Win32
    #include <windows.h>
#endif
```

#endif ends the #ifdef loop. #ifndef does the same thing, except if something is NOT defined. You will see how useful these are later, especially for debugging!

Coding conventions:

Every C++ programmer has his or her own “style.” For defining variables, the most popular ones include: myVariable and my_variable. I like the second one the best, because when I started out in game programming, I was using a popular program called Game Maker. The scripting was GML and most of the sample code I studied was using that style. So I got used to it. Pick whichever one you’d like.

For functions, use either MyFunction or my_function. For macros, aka “defines,” use ALL CAPS! Some people go even farther and use a letter at the beginning to name the “type.” If I was creating an object, I would put o_m16. If I were creating a sound effect, I would put sfx_m16. Just whatever you want, and whatever you think looks cool! There are many other coding conventions for variables, if you like none of them, don’t use them! Or think up your own. Coding conventions are used to your benefit, they keep you organized. This way you can use the same name (such as m16), except have it a different “type.” Theoretically, of course.

For function “block styling,” there are 4 or more possible popular styles:

```
int function()
{
    // 4 space indention, then code
}
```

The above is my favorite, and the one I use! Below is the “scripting” style:

```
int funtion() {
    // stuff goes here
}
```

The one above looks sweet to me, but it’s just not my favorite... The one below I think is ugly:

```
int function()
```

```
{  
  // stuff goes here  
}
```

The last most popular way is on one line. I use this style too, but only for short functions:

```
int function() { // stuff goes here }
```

Since it's so tiny, I hope you realize it would be best to make this function an **INLINE** function, since it makes the program the slightest bit faster, which is always a plus:

```
inline int function() { // stuff goes here }
```

Another small notation, it doesn't hurt to place semicolons at the end of a function, but it is not needed! So therefore, it's best to leave it off:

```
int function()  
{  
  // stuff  
}; // the semi-colon is not needed, but it doesn't hurt
```

In every file, it's good practice to keep a big comment up at the top of a source file. This way, other people can look at your code and know what that file does. Here's the way I do it, and I recommend you do it as well:

```
/*  
Program:      Example Program  
File:         main.cpp  
Function:     Main program loop and execution.  
Description:  Creates SDL window, initializes all variables,  
              and sets the FPS.  
Author:       Cam Winningham (LiceFork)  
Environment:  Any SDL supported platform.  
Notes:        Tested on Windows XP and Mac OSX Tiger (intel)  
License:      Custom, see "license.txt" in doc directory.  
Revisions:    1.00  4/7/06  Initial release  
              1.01  4/15/06 Added sound effects  
*/
```

Not only does it look professional, it's a lot easier for developers to know what's going on!

More information:

Well, by now you should have learned the basics of the C/C++ programming language. A wise man once said, "It takes one year to learn a new skill. It takes

three years to master it." From now on, keep coding. Keep coding until it makes you sick, then, in 3 years, you will be the next one writing the greatest and easiest tutorial series!

If you feel you didn't learn enough, or want to learn more, I'd highly recommend buying a copy of C++ for Dummies. That, my friend, is what started me out. Even after reading that, I was not satisfied. I read many... many countless articles, books, any tutorials on C++. I try to get the most out of it, and know for a fact that I can naturally write C++ programs.

Writing this is another challenge I put upon myself to quadruple check and make sure that I 100% know C/C++. It has been 5 years since I've learnt (and hopefully, especially by now, mastered) C/C++.

Conclusion to chapter 2:

Well this concludes the second chapter to my guide. Hopefully by now you have a good knowledge on how C++ works, and can make your own simple programs easily. There is a lot more you could possibly learn such as "the stack", command-line arguments, and object-oriented programming (which I will go over later), but I went over the extremely important and required need-to-know's in detail. I will teach you new tricks, as I need them, so don't think I'm just being lazy! Another reason for this is... I can't think of them all! In the next chapter, you will learn how to use SDL to make more complex programs!

Special thanks go to:

- Monster Energy XXL: for being a really huge energy drink that works, and tastes good!
- Coffee: 'nuff said!
- Angels Way: it's where I typed most of this up at ;)
- My Dad's Laptop: the way I typed it up away from home
- All NoResidue Fans: for giving me motivation
- Food: for keeping me alive
- The Internet: I would know nothing without it!
- Sam Lantinga: for inventing SDL!
- Bell Labs: for inventing C!
- Computers: for being so literal

Contact the Author:

You may contact me via e-mail: LiceFork@gmail.com

Also, check out my site for more tutorials: www.blackenedice.com

If you have AIM, my sn is LiceFork

Please leave any comments, good or bad, and tell me how my guide helped you out! (Or... destroyed your life)

Also... please report any found errors!

-Last updated: 4/15/06-