Review/Basic Memory/ Functions

Contents

Review – C++

C++ Hello World Code:

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main() {
        cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
        return 0;
} //main
```

```
#include <iostream>
```

- # means do first (before you compile the rest of the code)
- include means include this library or header in your program
- iostream – a library with input and output functions (cin, cout)

using namespace std

- means we don't have to specify the std first.
- E.g., If we didn't say "using namespace std", we'd have to specify std::cout instead of just cout

int main() {

- Every C++ (and C) program must have a main function.

- This function is run first and automatically.
- Like base camp. It's where you start, where you come back to, and where you end.

cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!

- cout: character out. You're piping characters into cout and they get printed, usually to the console window
- endl: new line. It also flushes out the buffer.

// comments.

Note that every line of code ends with a **;**

return 0;

CIN

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main() {
        string response;   // what is response?
        cout << "Please enter your name:" << endl;
        cin >> response;
        cout << "Your name is " << response;
        return 0;
} //main
```

Operators

+, -, *, /, %

C++ also allows you to do:

```
int x = 4;
x++;
x--;
```

Comparators and Operators:

```
== , != , < , <=, >, >=
&&, ||, !
=, +=, -=, /=, *=, %=,
```

/**/
FYI ONLY: Bitwise Operators (we can operate at the bit level):
(We most likely won't use these operators in this class.)

```
int A = 60;          //0011 1100
int B = 13;          //0000 1101
C = A&B;                    //0000 1100  Binary And Operator
C = A|B;             //0011 1101   Binary Or Operator
C = A^B;             //0011 0001          Binary Exclusive Or
C = ~A;              //1100 0011 Binary One's complement (flips all the bits)
C = A<<2             //1111 0000  Binary shift bits (shifts them to the left by 2)
C = A>>2             //0000 1111  Binary right-shift
```

Bit operators used in encryption algorithms, ports and socket communication (network stuff – checking parity, checksum, etc.), graphics, etc.
/**/

Comments:

C++ allows you to add comments in 2 different ways:

```
// works for one line only
// you'd have to put another // before each new comment line

/* Whereas you can put as much stuff and as many lines
   As you want with this type of comment.
   See what I mean?
*/

/* equally, you can put this around code with // in it.  For example, you could do this:
int main() {
        cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
        return 0;
}
   And you've just commented out the main function  (so nothing will work)
*/

/* One more note:
 * You often see blocked comments like this so that there's
 * no confusion that each line is still part of the comments.
 * It also looks nice.
 * There are a bunch of different variations of this format,
 * but almost all involve the star at the beginning of each
 * line within the block comment.
 * The thing is – it's inconvent to type.
*/
```

NOTE: in eclipse, you can highlight areas you want commented out and choose either
Source->toggle comments (to add or remove comments)
Or
Source->Add Block Comment (and Source->Remove Block Comment)

Note 2: I don't care whether you use block comments or in-line comments when you comment your code.

Functions:

```
int max3(int num1, int num2, int num3)
{       int result;
        result = max(max(num1,num2),num3);
        return result;
}  //max3
```

The c++ compiler must be made aware of functions before you can use them.

E.g., this won't work!!!!!!!:

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main() {
        int x = max3(4,7,3);  // problem
        cout << x << endl;
        return 0;
} //main

int max3(int num1, int num2, int num3)
{       int result;
```

```
result = max(max(num1,num2),num3);
return result;
} //max3
```

The function max3() is called in the main function, but there is no evidence that max3 exists if the compiler is working its way down from the top.

You must **declare** a function before calling it.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int max3(int k, int m, int n);  // Here is our function declaration

int main() {
          int x = max3(4,7,3);
          cout << x << endl;
          return 0;
} //main
int max3(int num1, int num2, int num3)   // Here is the function definition
{      int result=max(max(num1,num2),num3);
return result;
} //max3
```

- Declaration: The compiler can now handle most uses of the function without needing the full definition of the function.
- Declaring a function lets you write code the compiler can understand without all of the details.
    - Especially useful when one file is using functions in another file

VARIABLES:

What is a variable?

"a storage location (identified by a memory address) paired with an associated symbolic name (an identifier), which contains some known or unknown quantity of information referred to as a value."

wikipedia

Why do we specify the type?

- What we can do with the variable
- How much ***space*** to set aside in memory for the value

Example:

int x = 3;

- **3 parts:**
- **The name of the variable**
    - **x, y, ct, etc.;**
- **A location in memory**
    - **(you don't see that)**
    - **We need to know how much space to set aside in memory – that's what int/double/bool, etc tells us**
- **A value**
    - **Goes in that location in memory**

Memory

	3 X 0x32ef12			

int x = 3;
cout << x << endl;

- What prints, the name of the variable, the value of the variable, or the address of the variable?
- Most of the time this is what we want…
-

What if we want to get the address in memory instead of the value?

- cout << &x << endl; //will print out 0x32ef11 (address of x in memory)

POINTERS:

What if you want a variable that holds **an address** (as opposed to a value)?

```
int *y;  // y is a variable that holds the memory address of an integer.
```

Now y can hold an address. To give it an address, you could do either:

```
int x = 3;
int *y = &x;
```

Or

```
int x = 3;
int *y;
y = &x;
```

Note: Most of the time you want to deal with the values, not the addresses. If you want to use the value at the address that y holds, you'd use *y. So, for instance, to add 7 to the value at the address y holds, you'd do:

```
int z = *y + 7;
```

And to print out the value at y's address, you'd do:

```
cout<< "The value at the address that y holds is: "<< *y << endl;
```

If you want to see the actual address that y olds, you would do:

```
cout<< "The address that y holds is: "<< y << endl;
```

To make sure this is the memory address of where x is located, print out x's memory address:

```
cout<< "x's memory address is : "<< &x << endl;
```

Finally, to see where y is located in memory, you'd do the following:

```
cout<< "y's memory address is : "<< &y << endl;
```

If you are wondering why we'd even need a variable that holds an address, HOLD THAT THOUGHT! There are a bunch of reasons why we'd want to use a variable that holds an address. I promise that you will be using them and see their value throughout this course. For now, just learn how to use them and how they work.

It works the other way too. You can change the value at what y points to as follows:

```
*y = 42;
cout << x << endl;
```

FYI *y can be used as if it is an integer. It points to an integer, so you can use it as if it is an integer. It may look weird, but you can do this:

*y = *y**y;

& versus *

int x = 4;
// this:

a. sets aside an address in memory at a location large enough to hold an int
b. Names that location in memory x
c. Puts the integer 4 into that location

cout << x << endl; // prints out the value at location of x.

- Compiler assumes you want the value at that location, not the location address.

cout << &x << endl; // prints x's memory address

int y = &x; // can't do. Different types.

- The address of x is not an int, and y is automatically making a new address at a location in memory that will hold an int.

int *y = &x; //can do. Same types

- Says the variable y holds an address ***(aka is a pointer)***
- y now holds the address of x (x is a variable that holds an int)
- …or y **points to** a location that holds an int,

cout << y << " " << *y << endl;   //gives you 0x32ff1c 4

C++ Functions and parameters:

Discuss:

```
void whaddayathink(int x);
int main() {
	int x = 42;
	cout << "x is " << x << endl;  // What is printed here?
	whaddayathink(x);
	cout << "x is " << x << endl;   // What is printed here?
	return 0;
}
void whaddayathink(int x) {
	x = x + 2;
	cout << "x in whaddayathink: " << x << endl;    //what is printed here?
	return;
}
```

Now load the code and try it. What happened?

Why doesn't the value in x change after the call to waddayathink()?

The variable x in main is at a location in memory (0x32ef11)

- It holds the value 42.

When a function is called, ***a brand new parameter is created, with its own location in memory*** (0x4102cc)

- A copy of the value of 42 is placed inside that local parameter at location 0x4102cc
- That value is changed to 44 (so 0x4102cc holds the value 44)

But the value inside 0x32ef11 is never changed.

This is known as **CALL BY VALUE**

Call by value **is the default method for passing most data into functions**

Call by value means that **a copy** of the value is placed in the parameter.

Example (Try):

```
void changefunc (int j, int k);

int main() {
        int x = 7;
        int y = 3;
        changefunc (x,y);
        cout << x << endl;  //what gets printed?
        cout << y << endl;
        return 0;
} //main
void changefunc (int num1, int num2) {
        num1 = num1 * 10;
        num2 = num2 + 10;
        cout << num1 << endl;   //what gets printed?
        cout << num2 << endl;
        return;
} //changefunc
```

Works this way for all primitive types (including strings)

Call by value (summary):

This is the default when calling functions

- A copy of the variable's value is placed in the parameter
- If the parameter's value changes, it only changes within the function
  - because only the value in the address in memory associated with the parameter is changed, not the value at the address in the variable
- Outside the function, the original value remains unchanged.

Call by Pointer:

What if you actually want to change the parameter's values so that they remain changed outside the function (think of functions like swap or increment…)

Instead of sending in a copy of the value, we can send in the ADDRESS OF THE VARIABLE

```
int x = 42;

whaddayathink(&x);  // function call
```

Now if we change what is at the address, it will be changed both inside and outside the function

Problem: **`void whaddayathink(int x) {`**

- `The type of the parameter x is int`
- `NOT address of an int!`

The parameter's type must be a pointer!!!

```
void whaddayathink(int *x) {
	x = x + 2;
	cout << "x in whaddayathink: " << x << endl;
	return;
}
```

So the whole thing looks like:

```
void whaddayathink(int *x);
int main() {
	int x = 42;
	cout << "x is " << x << endl;
	whaddayathink(&x);
	cout << "x is " << x << endl;
	return 0;
}
void whaddayathink(int *x) {
	*x = *x + 2;
	cout << "x in whaddayathink: " << *x << endl;
	return;
}
```

Now, in main, the first cout prints 42, and the second prints out 44. The variable x has been changed inside whaddayathink().

Call by value vs. by pointers:

Call by Pointers:

- Can change variables from one function within the confines of another function
  - Makes code more readable
  - Reduces or eliminates the need for global variables
    - Better form (I was taught never to use global variables)
      - Easier to read
      - Don't have variables sticking around when you're done with them
- Smaller memory footprint
  - We're not making a new copy of each value passed into the function as a parameter.
  - (This applies more to arrays, structs, things bigger…)

`Why not always use Call by Pointer?`

Call by Value:

  - Can't accidentally change a value in another function
  - Make sure, if you're working with others, they don't accidentally change the value
    - Maintain "privacy"

Call by Reference

- ***Aka Aliasing***
- Variables are names we give to locations in memory (where we stick values). In call by reference, we're just giving a different, local name to ***THE SAME LOCATION IN MEMORY.***
  - So inside the function change3, the variable we named m in main is now named y inside change3
  - They're both the same location in memory, it just has a different name.
- ***Results are similar to Call by Pointer (the value remains changed outside of the function),***

```
void change3(int &x, int &y);
int main() {
	int k = 12;
	cout << &k << endl;  // gives us 0x61ff08
	int m = 4;
```

```
        change3(k,m);
        cout <<"k is now " << k << " m is now " << m << endl;
        Int q = 32;
        Int r = 18;
        Change3(q,r);
        return 0;
}
void change3 (int &x, int &y) {
//says that the address that we refer to as k is now also referred to as x
        cout << &x << endl;  // gives us 0x61ff08
        x = x + 3;
        y = y - 2;
        return;
}
```

Note: this is a c++ feature that is usually preferred in c++, but doesn't exist in c

swapped print, so you know when the values were swapped or not