C++ CLASS NOTES 6-11-14

**IF YOU ARE GOING TO COPY AND PASTE CODE, MAKE SURE THAT ALL VARIABLES AND FUNCTIONS ARE NOT MIXED WITH CAPITALIZATION…. WORD AUTO-CAPITALIZES SOMETIMES SO BEWARE**

**POINTERS**

**Reference operator (&):**  the address of a variable can be obtained by preceding the name of a variable with an ampersand (&), known as a *reference operator,* which can be literally translated as “address of”.

**Foo = &myvar;**

The actual address of a variable in memory cannot be known before runtime, but lets assume, in order to help clarify some concepts, that myvar is placed during runtime in the memory address 1776.

1. **Myvar = 25;**
2. **Foo = &myvar;**
3. **Bar = myvar;**

Terms:

**Pointer: attached to variable**

**Reference: to a memory location**

**Dereference (\*) : represent value of pointer**

\*: Used for content, and declare pointer. Without this asterisk in front of variable set with and ampersand for a pointer, the variable (even though it has the same var name, would change addresses)

You can also print the address of your pointer after runtime with cout << myPointerVarName; (this will give you the physical address where variable is stored) ex:1ff5

**Declaring pointers**

Due to the ability of a pointer to directly refer to the value that it points to, a pointer has different properties when it point to an int or a float. Once dereferenced, the type needs to be known. And for that, the declaration of a pointer needs to include the data type the pointer is going to point to.

The declaration of pointers follows this syntax:

**Type\*name;**

Where type is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

**int\*number;**

**char\*character;**

**double\*decimals;**

Note that the asterisk (\*) used when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator seen a bit earlier, but which is also written with an asterisk(\*). They are simply two difference things represented with the same sign.

**Example in Steps**

Int\*A; //declare as a pointer

myvar = 25;

A = &myvar //contains location value 1776

\*A = 10;

//new value of A is 10, but with same memory location (1776).

**Another Example in Steps**

{

Int firstvalue, secondvalue;

Int\*mypointer; //declare as pointer

Mypointer = &firstvalue; //firstvalue’s location is 1ff2

\*mypointer = 10; //firstvalue now equal’s 10, memory location is 1ff2 still

Mypointer = &secondvalue;//new memory location is 1ff9

\*mypointer = 20; //second value’s variable contains value of 20, but has location of 1ff9 because pointer has been moved to point to secondvalue, which occupies a different memory location (1ff9)

}

**Another Example in Steps**

**{**

Int firstvar = 5, secondvar = 15; //firstvar memory location is 0001, secondvar memory location is 0009

Int\*p1, \*p2; //pointers declared

p1 = &firstvar; //value = 5

p2 = &secondvar; //value = 15

\*p1 = 10; //value of firstvar (in location 0009) = 10 now

\*p2 = \*p1; //pointer1 now points to pointer 2 (0001 now points to 0009),

p1 = p2; // changes value of pointer, NOT Location in which it points

\*p1 = 20; // memory location 0009 value changes to 20, but not 0001, because p1 is no longer pointing to p1. Both p1 and p2 are now pointing to 0009, whereas before p1 was previously pointing to memory location 0001

**}**

**Pointers and arrays**

Int myarray[20];

Int\*mypointer;

The following assignment operation would be valid:

Mypointer = myarray;

AFTER that, mypointer and myarray would be equivalent and would have very similar properties. The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would not be valid:

**Example:**

Int myarray[5];

Int\*mypointer;

Mypointer = myarray;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-

{

Int numbers[5];

Int\*p;

p=numbers; \*p=10;

p++; \*p=20;

//the above method of adding to arraylocation with pointer is useful if you are populating the array from beginning index to end in order

//different way to do the same thing BELOW…

//the below method of adding to arraylocation with pointer is useful if you are populating the array from specific locations, or if you want to jump around

P= &number[2]; \*p=30;//this makes pointer point to first index

P=number+3; \*p=40;//number+3 shifts the new pointer location from the first index (redeclared above in p=&number[2]) by 3

P=numbers;\*(p+4)=50;

For(int n=0; n<5;n++)

{

Cout << number[i]; //displays output from entire array in loop

}

}

**Pointer Initialization**

**A pointer may be initialized to 0, NULL, or an address.** A pointer with the value 0 or NULL points to nothing and is known as a **null pointer.** Symbolic constant NULL is defined in header file <iostream> (and in several other standard library header files) to represent the value 0. Initializing a pointer to NULL is equivalent to initializing a pointer to 0. The value 0 is the only integer value that can be assigned directly to a pointer variable without casting the integer to a pointer type first.

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array):

**Int\*foo= &myvar;**

**Int\*bar= foo;**

**EXAMPLE:**

Int\*x = &y;// memory location of y is 1776, pointer x points to 1776

Int\*z = &x;//memory location of x, is now changed to 1776, because the pointer pointing to x is transferred to z, which is also in 1776 now. Also, the value of z is now y, with the same memory location of 1776.

**Pointer Arithmetic**

**Char\*mychar;**

**Short\*myshort;**

**Long\*mylong;**

**++mychar = 1 byte;**

**++myshort = 2 bytes;**

**++mylong = 4 bytes;**

\*p++ // same as \*(p++): increment pointer, and dereference unincremented address

\*++p // same as \*(++p): increment pointer, and dereference incremented address

++\*p // same as ++(\*p): dereference pointer, and increment the value it points to

(\*p)++ // dereference pointer, and post-increment the value it points to

**Example:**

Int x;

Int y = 10;

Const int \*p = &y; //memory location 0003

X = \*p; // value of previous y was 10, so value of x in location 0001is now 10

\*p= x; // **ERROR**, BECAUSE constant declared for pointer does not allow modification

**POINTERS AND const**

**Example:**

//pointers as arguments:

#include <iostream>

using namespace std;

void increment\_all(int\* start, int\*stop)

{

Int\* current = start;

While(current !=stop)

{

++(\*current); // increment value pointed

++current; // increment pointer

}

}

Void print\_all(const int\* start, const int\* stop)

{

Const int\* current = start;

While(current != stop)

{

Cout << \*current << ‘\n’;

++current; // increment pointer

}

}

Int main()

{

Int numbers[] = {10,20,30};

Increment\_all(numbers, numbers+3);

Print\_all(numbers, numbers+3);

Return 0;

}