**Pointers**

**Variable Anatomy**

int x = 5;

Value:**5**

**Label:** *x*  
**Type**: *int (4-bits)*  
**Memory Address** (*&x*) = *0x61ff1c*

Pointers are variables that store the address location of other variables. The purpose of a pointer for a stack memory variable, is to access a variable faster and more precisely than simple accessing it by name. For a heap memory variable, it is the only way to access it. A pointer is a variable in-itself, so it thus will also have it’s own unique memory address.

Value:

*0x61ff1c*

**Label:** *xpointer*  
**Type**: *int\* (4-bits)*  
**Memory Address** (*&xpointer*) = *0x61ff18*

**Pointer Anatomy**

int \*xpointer =&x;

**Reference Operator (&)**

The ‘&’ symbol will return the memory location of a variable

**Example:**

int x = 5;  
 cout << x; *// Outputs variable value (5)*

cout << &x; *// Outputs mem location (0x28fedc)*

**Dereference Operator (\*)**

The ‘\*’ symbol is used when first creating the pointer, and when you want to find the value of the variable the pointer is pointing to.

**Example:**

int x = 5; *// Variable initialization and value*

int\* xpointer = &x*; // Pointer initialization and value*

*//* ***“The value of the pointer = the mem location of x”*** cout << x; *// Outputs variable value (5)*

cout << xpointer; *// Outputs mem location of x (0x28fedc)*

cout << &xpointer; *// Outputs mem of xpointer (0x28fe18)*

cout << \*xpointer; *// Outputs x’s value (5)*

**Returning a pointer from a Function**

When you are creation a function that returns a pointer you must make sure that the return type is also a pointer.

**Example:**

int\* ReturnPointer(int x){

int \*y = x;

return y;   
 }

**Defining a Pointer Type a Name**

You can use *Typedef t*o define aliases for pointer type int\* and double\*. This means whenever you create a pointer, you’d name it with this new type you’ve created.

**Example:**

typedef int\* IntPtr; *// Creating variable int\* called IntPtr.*  
 IntPtr p; // Initializing pointer p

**Pointers with Arrays**

You can use pointers to point to arrays too! The pointer then becomes a pointer array itself. The difference though is that an array variable is still the size of the amount of elements in the array, while the pointer only points to one array element at a time, thus is the size of only one element.

**Example:**

**1**

**2**

**3**

**4**

**5**

**1**

Array a

\*p

20 bits

4 bits

int a[5] = {1,2,3,4,5};

int \*p = a;

cout << sizeof(a); // 20 bits

cout << sizeof(p); // 4 bits

**Memory**

Every process and variable created and run takes up memory space (RAM), and each has a specific location in which its accessed from. The goal of a program is not only for it to work, but it to work as effectively and efficiently as possible. Efficiency is measured by the amount of memory the program utilizes, (Being that you want to use the least amount of memory space as possible). While simpler code and smarter programming will cut down the amount of memory used, there are also different ways memory can be created, allocated, and accessed for a smoother experience.

**Stack:**

Stack is used for static memory allocation, as in the variables have their memory allocation pre-determined and stored right when the program is first compiled. This explains why Arrays must have a specific set size to them, as the stack cannot change. A literal explanation of how stack is made and why it can’t be changed is by referencing it’s name itself, how each block of memory gets stacked on top of the other, and to modify anything would require the stack to disassemble itself.

**Global:** functions and variables set in main are not equivalent to ones set globally. Ones set globally only need to be created once, and each time the global item is called upon, it uses the same memory location.

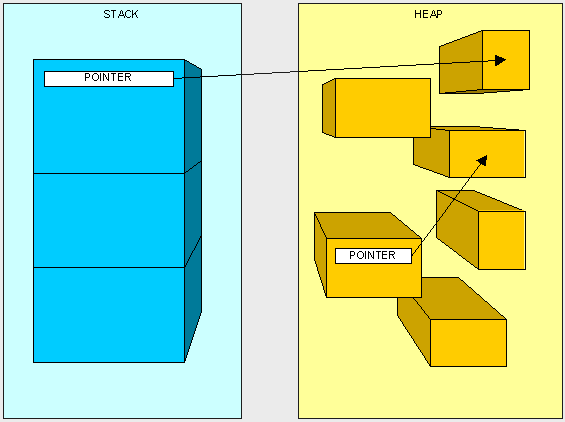
**Scope:** Scope is the code that is currently on the main stack. Since global variables and functions exist, not everything is always stacked. The stack has a preset amount it can handle, but as the code runs, the processes and memory utilization may still fluctuate. For example, when you call upon a global function, it gets added to the stack of processes and memory being used, however once the function is completed and it hits the final semi-colon, it removes itself and all its memory off the stack, and thus out of scope.

**Stack Overflow:** If the program desires more memory than the stack or the system can provide, the program, and possible the whole system, will crash. The stack could get overwhelmed by a having an excess of nested functions called (as the stack just continues stacking, and never removing the stacks it did not prepare for), or if there was an infinite recursive call. The system could get overwhelmed if it wasn’t prepared for enough RAM onboard to handle such a memory-intensive program.

**VS.**

**Heap:**

Heap is used for dynamic memory allocation. This is memory that doesn’t exist until its made. This gives the programmer a lot of flexibility when it is unclear how large or how many variables or objects he or she will need. This is an example of why arrays can be made without any size restrictions on the heap. Heap can be imagined as floating data, that though can only be accessed by it’s specific memory address. Since it is floating in the middle of nothingness and not set in a specific place of order like a variable on a stack, this makes stack a much faster way of retrieving data than heap.

 **Lifetime:** The lifetime of a heap variable is the amount of time it exists from the time it’s declared to the time it gets deleted, or when the program ends

**Memory Leak:** This is a failure to remove discarded memory. Heap variables do not deallocate themselves until the programmer specifically sets it to do so. A Memory Leak is most common to happen when a programmer accidentally looses track of the Heap variable’s memory address, by redefining it’s memory address pointer.

**Making Dynamic Variables**

- The "new" operator creates a dynamic variable on the heap.

**Example:**

int \*p = new int(5);

string \*s = new string(“Hello World!”);

*// ⤷ Creates a pointer that points to a heap variable*

cout << \*p; *// Outputs heap variable (5)*  cout << \*s; *// Outputs heap string (Hello World!)*

**Making Dynamic Arrays**

**Example:**

int \*a = new int[5];

**Making Dynamic Class Objects**

You can create a class object onto the Heap

**Example**

CLASS \*a = new CLASS(“Input member variables”);

\* Note: objects on the heap do not require operator overloads to perform operations onto it. \*

**Example 2**

CLASS \*a = new CLASS(); // Blank Input

(\*a).stringMember = “Test”

**\* Example 3 (USING *ARROW* OPERATOR)**

CLASS \*a = new CLASS(); // Blank Input

a -> stringMember = “Test”;

**Making Dynamic Class Members**

You can make class members onto the Heap

**Example**

class Example {

public:

Example(string n): name( new string(n) ){ }

string getName() const { return \*name; }

private:

string \*name;

**Shallow Copy**

A shallow copy copies the value of dynamic memory variables , and the two objects will share the same memory space for their data members.

Visual Example:

Example \*E1 = new Example(“Hi”) ;

// Has memory address 0xc0

Example \*E2 = new Example(“Bye”) ;

// Has memory address 0x20

**E1->shallowCopy(E2);**

E1 = “Hi” // Has memory address 0xc0

E2 = “Hi” // Has memory address 0xc0

**Example**

class Example {

public:

void shallowCopy (const Example\* e) {

this->name = e->name; }

private:

string \*name;

};

**Deep Copy**

A deep copy copies the values of the elements pointed to by dynamic variables, and this unlike the shallow copy, insures that the two objects have distinctly different memory spaces.

Visual Example:

Example \*E1 = new Example(“Hi”) ;

// Has memory address 0xc0

Example \*E2 = new Example(“Bye”) ;

// Has memory address 0x20

**E1->deepCopy(E2);**

E1 = “Hi” // Has memory address 0xc0

E2 = “Hi” // Has memory address 0x20

**Example**

class Example {

public:

void deepCopy (const Example\* e) {

\*(this->name) = \*(e->name); }

private:

string \*name;

} ;

\\

**The Big Three**

When using dynamic memory for class members, you must follow the big three functions that insure the validity and safety of the dynamic member.

**Copy Constructor**

While copy constructors are automatically generated without declaration, it features a **shallow copy**, when in fact we want a **deep copy**. This creates new and distinct memory space for an object.

**Example**

class Example {

public:

Example(const Example& e){

name = new string(\*(e.name));

number = e.number; }

private:

string \*name;

int number; };

**Destructor**

This function automatically deletes any dynamic objects go out of scope, thus preventing memory leaks.

**Example**

class Example {

public:

~Example(){

delete name; }

private:

string \*name;

**Assignment Operator Overload**

This operator allows us to return a reference to the object. The below example verifies that the current object is a different object, and if it isn’t, then it returns the reference to the object.

**Example**

class Example {

public:

Example& operator =(const Example& e){

if(this != &e) {

\*name = \*(e.name);

number = e.number;

}

return \*this; }

private:

string \*name;

int number; };

**‘This’ Pointer**

The this pointer is a pointer to the calling object.

**Example (Displaying Heap Object’s Memory Location)**

class Example {

public:

void displayThis() const {

cout << "this: " << this << endl; private:

string name;

};

main () {

Example \*a = new Example(“test”);

a->displayThis();

**Example 2 (Comparing Two Heap Objects)**

class Example {

public:

void compareObjects(Example\* object) const {

cout << object->name << endl; //outputs b value

cout << this->name << endl; // outputs a value

}

private:

string name;

};

main () {

Example \*a = new Example(“test”);

Example \*b = new Example(“test2”);

a->compareObjects(b); // a = this, b = object

}

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**Deleting a Heap Variable**

You can deallocate a heap variable using the delete function on its pointer.

**Example:**

int \*p = new int(5);

delete p;

This command does not delete the pointer, it just deallocates the heap variable, thus the pointer is now filled with a garbage value. To avoid confusion, you can NULL out the pointer.

**Example:**

int \*p = new int(5);

delete p;

p = nullptr;

**Deleting a Heap Array**

**Example:**

int \*p = new int[5];

delete [] p;