Class Five

You haven't run screaming yet... Let's do pointers!

- pointers are one of the most powerful things about c++
- A pointer is a variable that holds the address of another variable.

```
int *pnPtr; //a pointer to an int

int* pnPtr; //also valid

int * pnPtr; //also valid.

use whichever one makes sense to you.
```

- You can think of pointers as the address of your house. The address itself is just an address but it does tell someone how to come and find you.

- Pointers are addresses to objects - in this case your object would be your house.

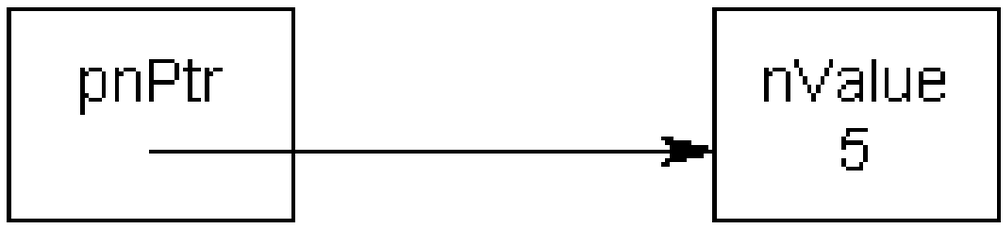
Here's an example of how they work:

```
int nValue = 5;
int *pnPtr = &nValue;
//assign address of nValue to pnPtr.
```

note: the * sign does not mean multiplication.

to get the address of a variable we can use the & (address-of operator)

Conceptually, you can think of it like this:




```
int nValue = 5;
int *pnPtr = &nValue;
//assign the address of nValue to a pointer

cout << &nValue << endl;
//print the address of variable nValue;
//0012FF7C would be the output

cout << pnPtr << endl;
//print the address that the pointer is holding
// 0012FF7C would be the output.
```

They are the same. Pointers much match the type of what they are pointing at.

Dereferencing pointers

The other operator you need to know about is *

You can dereference a pointer to access the object it is pointing at.

```
int nValue = 5;
cout << &nValue;
//prints out the address of nValue

cout << nValue;
//prints out the value stored in nValue

int *pnPtr = &nValue;
//assigns the address of nValue to the new pointer pnPtr

cout << pnPtr
//prints out the address value

cout << *pnPtr
//prints out the value at the address
```

Pointers can also be assigned and reassigned

```
int nValue1 = 5;
int nValue2 = 10;

int *pnPtr;
pnPtr = &nValue1;
cout << *pnPtr  //prints out 5

pnPtr = &nValue2;
cout << *pnPtr  //prints out 10
```

Null Pointers
You can set a pointer to = 0 when they are made. This is a really good practice to get into that will save you pain later. Never deference a null pointer. It will be very, very bad.

```
int *pnPtr = 0;
```

Note the size of a pointer is dependent on a computer's architecture, not on what it pointing at. They are all the same size. In a 64bit that is an 8bytes address. In a 32bit machine it's 4bytes.

Another way to think about it...

Pointers allow you to pass by reference, which means you pass the thing itself into a function. Just like references, this saves memory in a computer, particularly when dealing with large objects. In fact, a reference is a pointer that's simplicity deferenced.

Questions.

What is a pointer?

How do I create it?

How do I get to the object a pointer is pointing at?

Can they be reassigned?

?

If they are so similar, when do I use a pointer vs a reference??? To get there we need to get to a few things first.

And for what's behind the curtain...

Arrays are pointers that just point to the first element in an array! You can dereference it and it will return the 0 element of the array.

```
int anArray[5] = {1,2,3,4,5};
cout << *anArray  //this will print 1.
```

if you do this:

```
cout << anArray +1;
```

you will get address of the next element in an array via pointer arithmetic

to get the number, simply deference it

```
cout << *(anArray +1);
```

the () here say do this operation first then run the dereference, just like they mean in regular math. This is just the same as

```
cout << anArray[1];
```

Constant pointers.

A word to the wise. If you do not want people or yourself to accidentally access what you are pointing at and change it or to accidentally point to the wrong item, const is EXTREMELY helpful. How helpful? Sonic screw driver level helpful.

Dr Who's sonic screw driver often allows access to things or keeps others from accessing them when he's running away from bad guys.

A game example? You have built a game controller w/your arduino. You need your pointer to ALWAYS the same memory location on a chip because it's defined in a hardware spec for you.

Also, it protects your assumptions about your code base and allows a compiler to optimize code that it knows will not change.

This of this as **defensive programming. You will be alert for possible disasters**

Constant Pointer

A constant pointer will always point at exactly the same thing while the value itself can be changed

```
int nValue =5;
int *const pnPtr = &nValue;
*pnPtr = 6;
// but the address itself will NEVER change. You are just changing what's
//hanging out in nValue
```

int nValue =5;
int *const pnPtr = &nValue;
*pnPtr = 6;
// but the address itself will NEVER change. You are just changing what's
//hanging out in nValue

Pointer to a constant value

You access the value of the thing being pointed to, but not change it.

```
int nValue = 3;
const int *pnPtr = &nValue;
```

NOTE! You can still access the value here thorugh the value name itself
nValue = 8; //totes valid.

Also, the pointer itself can be freely reassigned.

```
int nValue = 5;
int nValue2 =10;

const int *pnPtr = &nValue;
pnPtr = & nValue 2;
```

Const pointer to a const int.
Can never change the value. Will never point at anything else - ever.

```
const int nValue - 5;
const int *const pnPtr = &nValue.
```

References vs Pointers + Functions

```
const reference
int val = 3;
const int &rVal = val;
```

Great! Why? Gives you access to an object but assures you that it can't be changed in the function. If you don't need to change something in a function, do it this way.

References act as const pointers that are implicitly dereferenced.

```
int nValue = 5;
int *const pnVal = &nValue;
int &refVal = nValue;

cout << *pnVal; //will return 5
cout << refVal //will return 5
```

Questions

What is a pointer?

Why make a pointer const?

Why make a pointer value const?

How do you make a pointer const and a value const?

Why make something const ever?

What is a reference?

Pointers and new.

c++ requires all pointers be declared by compile time. This is troublesome because what if we don't know what value something should be? AKA User inputs....

We use the new operator to tell the computer to save space for what we plan to tell it later on at runtime.

This is called Dynamic Memory Allocation

```
int *pnVal = new int;

you can then deference them to assign values

*pnVal = 7;

You can also do this with arrays too
int nSize = 10;
int *pnArray = new int[nSize];
```

Note we use new[ ] and delete [ ] with arrays.

You can also delete new objects. (more on this in a bit but for now note that will arrays you must use it he delete[ ] operator or you will get a leak.)

new is dangerous. You can leak memory here, which is DISASTROUS and will slow your game to a crawl over time.

```
void doSomething() {
   int *pnVal = new int;
}
```

Why is this bad? (hint... zombies.)

```
int nVal = 5;
int *pnVal = new int;
pnVal = &nVal;

Why is this bad? (losing keys to your apartment)
```

```
int *pnValue = new int;
*pnValue = 0;

delete pnValue; // this is how we delete a regular old pointer
pnValue = 0;

//why do this? Because pnValue would otherwise be freed up and would point god
knows where. Why is that trouble?

if(pnValue){
//do some stuff
}

//EVALUATES TO TRUE AS A RESULT OTHERWISE!!! If it's 0, it will be false
//and not run.
```

When to use pointers vs references?

Because pointers always point to a valid object it's impossible to reference deallocated memory. AKA no dynamic memory here people.

If it requires the keyword new (dynamically allocated memory) use a pointer -

For everything else, use a reference. It's safer.

How to use functions that pass by reference:

```
int nValue = 10;
int &refNVal = nValue;

foo(refNVal);

void foo(int &x)
{
   x = 6;
}
```

```
//Function call:

foo(&nVal);
or
foo(pntrValue);

//Function:
void foo(int *pValue)
{
    *pValue = 6;
 }
```

Remember arrays are just pointers and to pass them as such.

```
void printArray(int *pntrArray, int length) {
    for(int i=0; i< length; i++)
        {
            cout << pntrArray[4] << endl;
        }
}
```

Methods of an object

You can access all of an objects methods via a pointer using the -> arrow operator .

```
string myString = “hi!”
string* ptrToString =&mystring
ptrToString->size();
```

Returning pointers and references

Specify you are going to return a pointer and make sure you do that.

```
string* ptrToElement(vector<string>* const pVec, int i) {

   //returns address of the string in position i of vector that pVec points to
   return &((*pVec)[i]);
}
```

A word about scope.

```
//don't do this because if you return a pointer to a variable from a function
that only lives there it goes out of scope. This is a dangling pointer.

int* myFunction() {

    int myInt = 6;
    int *myPtr = &myInt;
    return myInt;

}
```

Questions:

1.Does a pointer always have to point to an existing variable?

2. Why store the address of an object that already exists?

3. How is a pointer different than the object it points to?

4. Can I dereference a null pointer?

5. What is a dangling pointer and why is it dangerous?

6. When would I use a pointer over a reference?

Homework:

Rewrite the homework from last week but now the functions should now take and return either references or pointers