**Overview**

**Java Memory vs C++ Memory**

Java and C# have a very different way of handling memory management than C++. Both Java and C# have a garbage collector; it essentially finds items in memory than are no longer referenced by anything in the main program and delete them. This provides users with a good hands-off experience of object management. They create an object, and when they are done the system cleans up after the object is no longer in use. There is the obvious problem in that the programmer has no real direct control as to when an object is removed from memory, but in general it is more than sufficient.

In C++ it is a lot more hands on with certain objects. There is generally an easy way to determine if an object needs manual management in creation of deletion and whether you can leave it alone. If the item is a Pointer, you need to create and destroy the object manually. We will cover pointers in the next section. If it is a normal object, then it will automatically be destroyed and removed from memory when it goes out of scope. For example, the following int requires no additional management from the user:



However, the following is a pointer to an int and requires the user to manually create and destroy:



We will cover pointers in just a minute, but first we need to look at the management of normal items in the program.

**Out of Scope**

Out of scope deletion is for normal objects. Essentially, anything that is not created by the user using the ‘new’ keyword does not have to be deleted by the user. Most objects, such as the int above, will be destroyed when the item goes out of scope.

To understand what out of scope is, let’s look at the following function:



The first line creates a double called **‘result’**. This is assigned the value of the argument called **‘input’** divided by 2. We return the result which will end the function. When the function is finished, it returns the VALUE of **‘result’** to the program, and the item **‘result’** itself is destroyed. In this case **‘result’** has gone out of scope. When the program leaves a function, all variables created inside that function are destroyed.

If we wanted to create a variable that will last and persist throughout the lifetime of a program, you can always create it in the main, or declare it in global namespace like below:



The top int ‘i\_will\_persist’ is created in global namespace and will exist within memory so long as the program runs. The other, ‘exist\_as\_long\_as\_main’ is created in the main function. When the main finishes, the program finishes, so it too will exist so long as the program runs.

Note: The names for variables here are really, really bad so try to use good names for everything you make.

**Pointers**

Pointers are a very powerful tool in C++ but they can also be very difficult to deal with and manage properly. A pointer is a variable that tells the program where an item in memory is located. As an example, a normal int describes a 32-bit whole number, whereas a pointer to an int describes where in memory that int is located. Run the following code to see:



To create a pointer to an object, notice we have added a \* to the end of the type declaration. To make a pointer to an int, we type int\*. To make a pointer to a double, we type double\*. To make a pointer to a CyberPet, we type CyberPet\*.

We also assign the pointer to point towards an already created int. We can use & to get the location of an item in memory. Appending & before the name of a variable returns its location in memory for us to use. Stating **&a\_number** is telling the program that we want to get the location in memory where **‘a\_number’** is.

We can also access the data that the pointer points to. Replace the line below:



With:



Notice how both ‘a\_number’ and ‘\*pointer\_to\_int’ print the same thing. Placing a \* at the start of a pointer gets the value of the item that the pointer points to. This can get quite confusing by adding \* all over your code, but it’s a useful tip to know.

So, why use pointers if a normal type can do the same thing? Passing the memory location of an object can in a lot of cases be faster and more efficient than passing the entire object. If we pass a class as an argument to a function, then the program has to manage the size of the class, whereas a pointer can represented by a single 32-bit number. Similarly, by passing a memory location to an object, any changes we make will affect the original as it points to only one object.

**Creating an item ‘On the Stack’**

To create an item dynamically, we need to become comfortable with two C++ keywords; new and delete. new is used to create an object in memory and gives the program a pointer to the object. delete does the opposite, it destroys the object that a pointer points to. For example:



The above code declares a pointer to an int called **‘pointer’**. We use the new keyword as a pointer is not the object, it just points to the object in memory. new creates the int and assigns the location to our pointer. We then assign a value to the int, print it out along with the memory address.

You may have noticed that there is no delete keyword used in this code. Technically, we have created an int in memory but we are not properly cleaning it up. This creates a ‘memory leak’ in our program. When the program finishes, the pointer goes out of scope and is destroyed but the int it points to does not. Because of this we have an object in memory that we cannot access again (unless you are unfathomably lucky with assigning a random address and getting it right), we can never properly delete it. Now the program does end at this point so it’s not the end of the world but it is still very bad practice to leave an object floating in memory like we are doing. The general rule to follow with new is:

*“For every new, there must be a corresponding delete.”*

Add the following to the end of our main so we properly destroy the int we have created:



**Creating and Destroying a Class**

The exact same syntax can be used to create and destroy a class.



As we are working with pointers and classes, there is another nice feature we can make use of. To get the value of a pointer, we append \* to the beginning of the object but with classes we have to wrap the whole thing in brackets. This whole process can be replaced with ->. Try amending the code above, replacing:



**References**

When you pass a variable into a function in C++, you don’t pass the actual object. Instead, a local copy is made in the target function which is then destroyed when the function finishes. This is where pointers can be very useful as we can create a pointer and pass it into a function, a local copy is made, and any changes we made to it will affect the original.

However if you create an object that is not a pointer and you want to pass it to a function where you can make changes to it, then we can pass it as a reference. A reference is a simple way of passing the original data into a function, it works similarly to a pointer in that it hands the memory address of the object to the function, but keeps the programmer a layer abstracted from all the difficult pointer syntax we would have to deal with.

The syntax for passing by reference is quite easy. If you want to pass the original object into a function, append a & to the beginning of the argument name. For example:



The above code passes an int into a function through referencing. Any changes made to **‘arg’** will also affect **‘an\_integer’** in the main. You can do the same thing with classes, so you can pass a class you want to make changes to without using pointers, you can pass pointers themselves by reference if you want to change what it points to, etc.

In Java and C#, everything is passed by reference with the exception of certain base types such as ints, and floats. In C++, we have to manually state that we want to pass by reference; otherwise it will just make a copy in the function we are calling.

**Exercises**

1. Create a class to describe a Vehicle. We need to know its registration number, the length of the Vehicle, how many wheels it has, and how many seats it has. Made sure that we have get/set function to read and change these values.
2. If you haven’t already, try and give the class a couple of constructors so we can assign values to the member variables when we create an instance of the class.
3. Create an instance of the class using a pointer.
4. Create a program that allows users to create, delete, and change all the values inside the class.
5. Create a function in the class that prints all the values onto the screen.

**Summary**

* To create a pointer to an object, add \* to the end of the declaration.
* Pointers are objects that point to the memory address of other objects of a specific type.
* Remember that for every instance of new, there must be a corresponding delete.
* new will create a new instance of an object and return a pointer to that object
* delete will destroy an instance of an object that a pointer points to. It will call the Destructor of that object (if it has one)
* You remove all the pointers to an object in memory and do not delete it (destroy the pointers but not what they point to) it will be virtually impossible to find again, meaning you can never free up the memory for use (until you close the program). This is called a memory leak.
* Referencing is a good way to pass an object to a function or class without using pointers.
* Adding & to the beginning of an instance will return the memory address of that instance. This allows you to save it as a pointer.



* To access the object a pointer points to, add \* to the beginning of the instance.

