Lecture 32

Intro to Memory Management

Java and C++ both store variables on the stack. When a new method/function is started (e.g., `main`), space is allocated on the stack for any local variables that are declared in that method.

Sample Java code:

```
int x = 10;
float a = 2.4713;
MyClass m;
```

In Java, there are two types of values – primitives and objects. Primitives are stored directly on the JVM stack. When we declare and object, we are really creating a reference variable that holds the memory address of the object.

In java, if we just declare the reference, we get the name, but not the object (because it hasn't actually been created). Remember that if we don't initialize a value, Java will do it for us. Java references are always initialized to `null`.

When we use the `new` operator in Java, we are telling Java to create a new instance of an object. It will be created on the *heap*, which is just memory that the program has access to that is not on the stack.

```
m = new MyClass();
```

creates the object on the heap, and assigns the heap memory address to the reference variable `m`.

We can create another object of the same type.

```
MyClass m2 = new MyClass();
```

Which creates it's own instance. That is, `m` and `m2` point to different objects.

We can reassign either of these to other already created objects of an appropriate type.

```
m2 = m;
```

Notice that `m2` and `m` now point to the same object. Also notice that the object `m2` originally pointed to is no longer referenced any where. In Java, the *garbage collector* is a separate process that may recognize that there are no references, and reclaim the memory to be used later.

C++ is a little different. The default is that everything behaves like a primitive, and so even objects are stored on the system stack.

Thus, if we write similar code to that above, we get very different results.

The primitive `int` and `float` values are handled the same way as in Java. The actual value is saved to the system stack. But in C++, objects of a class are, by default, also saved to the system stack.

When we say something like

```
MyClass m2 = m;
```

We create a new object on the stack, taking up as much space as needed, then we copy over the values of `m` into `m2`. In C++, the default is that the variable `m2` won't "point" to the object `m`, like it does in Java.

Of course, this is C++. We can do what we did in Java, we just need to learn a little more syntax.

C++ references and pointers

C++ actually has two mechanisms that are similar to Java references. They behave in slightly different ways, but both are worth knowing about.

First, we'll talk about C++ *references*. A reference is similar to a Java reference, except that it can only ever point to one thing. We can declare a reference variable as follows:

```
int i = 4;
int &ri = i;
```

Now, `ri` is effectively just another name for `i`. If `ri` changes, then `i` changes, and vice versa. Note that because it only ever refers to one thing, it must be initialized when it is declared.

References are very handy as formal parameters in functions.

See the sample code `swap.cpp` from the code page for this lecture.

Here, we've written a swap routine that is intended to swap the value of two integers. Of course, it doesn't work because we make a copy of the integers `x` and `y` when we call the function.

If we use references instead, then we are essentially copying the address of the integers `x` and `y`, so that changes in the function are reflected in the actual parameters. That is, `val1` and `val2` are other names for `x` and `y`, respectively.

We talked last week a little bit about how much copying C++ does when a function is called with an object parameter. The solution to this is to pass the object by reference, not by value.

See the `vector_ref.cpp` code from the code page for this lecture.

Remember, that the last time we looked at this code, we were just passing the vector to the max function, and max function made a copy of the vector.

This was bad, because we had to copy the whole vector (an $O(n)$ operation) before we even start to search for the max value. But it was potentially good, because it guaranteed that the value of `nums` in the calling function wouldn't be changed. We may want this.

We can pass the vector as a reference by making the formal parameter a reference variable declaration instead of a regular vector declaration. Now, we are only copying the address of the vector.

However, since the name `nums` in the `max` function is just an alias, changes we make to `nums` in `max` are seen when the function returns.

As we said, this may not be preferable. The solution is to label the formal parameter declaration as `const`. In C++, this means that we cannot change the variable, **or its underlying object**.

The `max` function works fine if we don't set a new value. How does C++ know? What's going on?

C++ pointers

C++ also has data types known as *pointers*. These are actually more like Java references than C++ references are, because they can be assigned over and over again to different objects of an appropriate type.

We declare pointers with respect to the type that they point to, and we use the '*' to indicate that the variable is a pointer.

```
int *ptr;
int x = 5;
int y = 7;
```

While it looks like we've named our first variable `*ptr`, this is not the case. You can think of this declaration more like:

```
(int *) ptr;
```

where we have a variable named `ptr` of type `int *`.

Like a reference, a pointer is actually a memory address. Unlike a reference, when we use the variable name, we are referring to the address, not the value. We can set our pointer to "point" to the value stored in `x` as follows:

```
ptr = &x;
```

The & is not the same as the reference indicator we just saw. In this context, it is the *unary address-of operator*. When placed in front of a variable, it returns that address in memory where that variable is stored. Above, "`ptr`" is an `int` pointer, and we are setting its value to the address in memory associated with the name `x`.

If we print the value of `ptr`, we will get the memory address. If we want the value that `ptr` points to, we need to use the *unary dereferencing operator* *.

```
int z = *ptr;
```

`z` now holds the value pointed to by `ptr`, which was also the value stored in `x`. That is, `z` holds the value 5.

Since declaring a pointer is just creating a name that can point to an address, we must have an object to point to. We can create a new object in C++ in the same way that we create a new object in Java.

```
vector<int> *vptr = new vector<int>;
```

See sample file `ptr_exs.cpp` on the code page for this lecture.

Looking at the function `objectPointers`, we create a pointer to a vector, then create a new vector object to associate with it. This is much more like the Java code that we are used to.

If we want to access members of the vector class, we need to dereference the pointer (so that we are referring to the object, not the address). We have to put the variable and dereference operator in parentheses because of the precedence of C++ operators. That is, if we leave the parentheses off, C++ will try execute the "." operator before the "*" operator.

We add the integer values 0 through 9 to the back of the vector by dereferencing the pointer, then calling the object's `push_back` method.

We print out the value of the pointer, and we get a large number. This is just the address of the vector object in main memory. Then we print out the members of the vector. Again, we have to dereference first.

This dereferencing is somewhat awkward, so C++ gives us a special operator to make it easier: "`->`".

The method `objectPointersBetterApproach` is essentially the same method (without the prints of the address values) as `objectPointers`, except that it uses this new, simple operator.

One important thing to note about these methods. In Java, this code would be fine, because the Java Garbage Collector could clean up the vector we created after the method completed.

In C++, there is no default garbage collector, so when we call the `new` operator, we are allocating memory. It is **our responsibility** to clean up after ourselves! In our example, it doesn't really matter, because after the function completes, the program itself completes. But imagine that your program is actually a web server that is supposed to be up for weeks or months at a time. Then every time we call the `new` command, you are allocating memory. If you don't tell C++ to reclaim it, it will remain unusable for any other methods that need it.

Now, imagine that this is a function that is called over and over. Maybe every time a web page is requested. Then we are just taking memory, and taking memory. This is known in programming as a *memory leak*, and it can cause long running programs to crash due to out of memory errors. It can also slow down performance because C++ has to keep track of all the allocated memory, so it's tables will grow and grow. These bugs can be very difficult to track down, because they don't occur close to the code that caused them.

The function `objectPointersBestApproach` shows the use of the `delete` command that deletes the object we created with the `new` command.