Functions, Methods & Arguments

Last class, we talked about I/O, and in particular the << and >> operators. They can be used to read or print any primitive data type, and later we will see that they can be extended to read and print class instances (i.e. objects), too.

Section 2.2 of Weiss also introduced strings and arrays. Syntactically, can any one tell me what strings and vectors are? They are templated classes. Classes you are used to. The string class in C++ is similar to the string class in Java, although unlike a Java string you can modify a C++ string. The vector class is similar to the ArrayList class.

We will introduce templates later (trust me). But for the moment, know that the angle brackets in the vector declaration contains a data type, and the vector is only allowed to hold instances of that data type.

OK, let’s get to our main topic: function calling and parameter (argument) passing. And let’s start with the language you know, Java.

If I am writing a Quagga class is Java, I might write a method called Grow(); It takes no arguments, but internally it multiplies the weight field of the quagga by 1.1 and then returns the new weight. Now, if I call Grow() on an instance of a Quagga, what happens? The Quagga grows! The weight field of the Quagga is permanently changed by the method call.

Why? What is happening “under the hood”? How exactly does the weight of the Quagga change?

Two implementation mechanisms you need to know: (1) method calls hide arguments. How did the function implemented as the Java Grow() method even have access to the weight field of the quagga? Or, equivalently, what is the difference between a function call and a method call?

Answer: when you call a method, the class instance you call it on is passed as a hidden argument to the function. For example,

Quagga q;
q.grow();

Can be written as a function call instead of a method. If grow were a function, it would be called like:

Quagga q;
grow(q);

The only difference between a method call and a function call is a hidden class instance argument. In fact, this is how methods are implemented: as functions with an extra, hidden argument!
In java, you have no functions, only methods, so there is always a hidden argument (well, unless you use static). In C++, you can write methods or stand alone functions, your choice.

When to use which? Well, the key is encapsulation. Keep code and the data it operates on close together. Therefore if you are going to change data in a class instance, make it a method of that class instance.

When to use a function? Two cases: (1) a general-purpose function that does not side effect anything (no changes to memory other than the return value), and (2) code that changes two or more things (passed in as arguments) symmetrically (harder to judge).

Now, back to the Quagga. Why does the quagga change when I call the grow method, if what is really happening is a function call with the quagga as an argument? Because of implementation method (2): Java is call by reference (mostly; see below). The quagga inside the function call is just a reference to the quagga object that exists in the calling object. The data is not copied.

So what happens if I write the following method?

```cpp
bool Quagga::foo(int i) { i *= 1.1; return true;}
```

Does the integer $i$ change? No!

Why not? Java isn’t always call by reference. It is call by reference if the argument in an object, but call by value if the object is a primitive. It copies the primitive and hands the copy to the function. The integer in the calling function is never changed. Huh?

That’s right: Java is call by reference for objects, call by value for primitives.

Why? Because primitives aren’t really objects in Java. But this makes sense. Most of the time, the goal is not to side effect a primitive, and making copies of every primitive would be expensive. (Java pretends not to care about speed, but it is inconsistent). Copying objects is expensive, however, and in an object-oriented framework methods want to modify their instances, so most of the time, call by reference is good. Java simply casts these rules of thumb in stone.

Why teach you about function calls in Java? Because C++ gives you more options. In Java, objects are copied when passed as arguments, primitives aren’t. In C++, you can pass any argument by either method. Let’s assume I write Grow(q) as a function (even though a method would be better style). If I want to modify the instance of the Quagga class, I write:

```cpp
void grow(Quagga& q)
{
    q.weight = q.weight * 1.1;
}
```

This assumes that Quagga is a class and q.weight is public. (One of the reasons that a method is better style is that weight doesn’t have to be public.)

The & after Quagga says pass this argument as a reference, i.e. don’t copy it.
If I leave off the &, then the quagga will passed by value, i.e. copied. This means that the copy of Quagga would have its weight changed, but not the original version (i.e., a bug).

So mechanically, the difference between a function and a method is that a method is a function with an extra hidden argument, the class instance.

A little terminology. When a value is passed in by reference and the method or function modifies it this is called a side effect. Why?

If an argument is passed by reference, is it changed? In general you don’t know. At best, you could go look at the implementation (cpp) file, but (a) you don’t want to have to do this, and (2) it might not be easy to know (if that function calls another using the same argument…). Java tries to remedy this problem through documentation (Java Docs), but you don’t really know.

If an argument is passed by value, you are safe: it won’t be changed as a side effect. If it passed by reference, you aren’t sure.