Scope

Last class, we discussed how methods having hidden arguments and the difference between pass by value parameters and pass by reference parameters.

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, you have to go get that functions source code… Java tries to remedy this problem through documentation (Java Docs), but the documentation may not match the source code, so 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.

But what about efficiency? Well, for primitive data types (int, double, etc.) it doesn’t matter: passing a reference (usually, although not necessarily, implemented by passing an address) is just as expensive as copying a value.

But for complex objects, copying the whole object may waste both memory and cycles. Imagine a large object, like an image or a video or a database. Call by reference is faster for big objects.

There is an efficiency down side to call by reference, however. Let’s say I have a complex object with lots of fields (quaggas have weights, heights, partners, ages, etc.). I may be processing that data, and some of it may already be in registers. Now I make a function call, and pass the quagga by reference. What happens to the values in registers? (Assuming the called function didn’t need all the registers) After the function call, the compiler has to assume that the function call changed them, and that they are no longer valid. As a result, they have to get re-fetched.

So C++ has a better option for big objects: constant call by reference. Pass the argument as a constant reference. This is done by putting const before the argument, and & after it. Now the object doesn’t have to be copied, but user’s know that it isn’t being changed, and so does the compiler. Best of all worlds.

So in summary, there are three ways to pass an argument in C++:

1. Call by value. Should be your default method for primitive data types.
2. Call by reference. Should only be used when you intend to side-effect the argument.

By way of comparison, Java always passes primitives by value and objects by reference. Java does not have a call by constant reference option.

In C++, any parameter can be passed by any of the three mechanisms above, whether it’s a primitive data type or an object. Most of the time, you should pass primitives by value, unless you want to side-effect them (in which case you use pass by reference). Similarly, most of the time you should pass objects by constant reference, unless you want the side-effect, in which case you remove the const term and just use call by reference.
Remember, too, from previous lectures that methods have a hidden argument that is the object the method is called on. This hidden argument is by definition an object and is passed by reference as a default. You can put `const` after the parameter list to pass the hidden argument by constant reference. (There is no way to pass the hidden argument by value.)

A new term you need to know is *scoping*. Scoping refers to the lifetime of a variable. When is it created, when is it destroyed, and when it can be accessed it in between. I will particularly focus on creation and destruction.

In C++, scoping is intimately related to memory location. More specifically, it is related to whether a variable lives on the stack or in the heap.

A quick review from CS270 (or its equivalent): Every process has a virtual memory space. Part of this is reserved for the operating system. The rest is divided into stack and heap. The stack starts at one end (typically the top), the heap at the other. The first function is always ‘main’, so the first frame is main’s frame. Main calls functions, which create new frames, which extend the stack.

What is in a frame? Some stuff the compiler uses to manage function calls (the program ctr, slots for register values, etc.). As a programmer you can ignore this and assume it works. But what you need to remember is that local variables are stored on the stack, in the frame of the function they are a part of.

The scoping rule for local variables: they are allocated when the line that declares them is executed, and they go away when the function returns (or, more accurately, at the end of the block they are defined in, as signaled by `}`).

The scoping rule for parameters: parameters are allocated when the method of function is entered, and they leave scope when the function returns.

Scoping is critical because constructors are invoked anytime an object enters scope, and destructors (which Java doesn’t have) are invoked when an object leaves scope. Therefore you need to know about scoping.

But be precise: constructors and destructors are called when an *object* enters or leaves scope, but not when a *reference to an object* enters or leaves scope. It is important to distinguish between primitives, objects and references.