Functions, Pointers, and the Basics of C++ Classes

William E. Skeith III

Functions in C++

Vocabulary

You should be familiar with all of the following terms already, but if not, you will be after today.

- function prototype
- function header
- function body
- call by value
- call by reference
- actual parameter (the view from outside the function)
- formal parameter (the view from inside the function)
- default parameters
- function overloading

By-Value vs. By-Reference

When parameters are passed *by-value*, the formal parameters of a function refer to *copies* of the actual parameters which are discarded as soon as the function call returns. In *by-reference* calls on the other hand, the formal parameters become synonyms (or aliases, if you will) of the actual parameters. So, changes to by-reference parameters are of course reflected in the actual parameters. To completely drive the point home, let's look at a few examples. Consider the following function that increments the formal parameter, which is passed by value.

```
void addOne(long x) {
    x++;
    return;
}

int main(void) {
    long y = 23;
    cout << "y == " << y << endl; //y contains 23.
    addOne(y);
    cout << "y == " << y << endl; //y is unchanged: still contains 23.
    return 0;
}
```

In this case the formal parameter `x` is a local copy of the actual parameter `y`. The lifetime of `x` is only that of the function call, so certainly the changes will not be reflected outside the scope of the function. Now let's look at a similar example using by-reference parameters.

```
void addOneByRef(long& x) {
    x++;
    return;
}

int main() {
    long y = 23;
    cout << "y == " << y << endl; //y contains 23.
    addOneByRef(y);
    cout << "y == " << y << endl; //y now contains 24.
    return 0;
}
```

Again, since the formal parameter `x` is a synonym for the actual parameter `y`, changing `x` will of course change `y`. Note: *unless you explicitly instruct for parameters to be passed by reference, they are always passed by value in C++.* It is natural for us as humans to condense incoming information, describing "new" ideas in terms of old ones, and only accepting new concepts when necessary. In an attempt to speed up the process, consider the following example which does away with by-reference parameters altogether. Suppose you have a function that takes a by-reference parameter of some type- it doesn't matter what. For now, we'll stick with `long` although it is irrelevant. The function may look something like this:

```
void someFunction(long& x) {
    x = x+7;
    //other stuff...
}
```

We can make a function that is equivalent in almost every way with pointers as follows:

```
void someFunction(long* px) {
    long& x = (*px); //explicitly alias x to be what px points to.
    //the rest of the code is the same:
    x = x+7;
    //other stuff...
}
```

This will change the way that you must call the function in C++ (you'll have to send `&x` rather than just `x`), but beyond that, they should be basically identical. Here's `gcc`'s assembly output (with optimization enabled) for the two versions:

```
# version that uses call by reference:
_Z12someFunctionRl:
.LFB30:
        .cfi_startproc
        addq        $7, (%rdi)        #, *x_2(D)
        ret
        .cfi_endproc

# version that passed a pointer by value:
_Z13someFunction2Pl:
.LFB31:
        .cfi_startproc
        addq        $7, (%rdi)        #, *px_1(D)
        ret
        .cfi_endproc
```

Note that they are identical. Now consider the following example:

```
void mulByTwo(long A[], unsigned long n) {
    for(unsigned long i=0; i<n; i++)
        A[i] = A[i]*2;
    return;
}

int main() {
    long Arr[10];
    unsigned long i;
    for(i=0; i<10; i++)
        Arr[i] = i;

    for(i=0; i<10; i++)
        cout << Arr[i] << "  "; //will print 0...9
    cout << endl;
    mulByTwo(Arr,10);
    for(i=0; i<10; i++)
        cout << Arr[i] << "  "; //will print 0,2,...,18
    cout << endl;

    return 0;
}
```

At this point, you may be thinking, "Hey, I didn't see any ampersands! I thought you just told me everything was by-value unless I said otherwise?" to which I would respond, "Yes I did, but only because it's true." The confusion you may be experiencing is simply due to a failure to see C++ arrays for what they really are: arrays are just pointers. So, an array parameter like `long A[]` is actually a pointer: `long* A`, and that pointer *is* passed by-value.[1] If you were to change the *value of the pointer* from inside the function, it would not change anywhere else. But if you dereference that address and change what's in memory, of course that change will persist. There's only one array, but you now have two copies of its address. In fact, I would encourage you to not use the bracket notation (`[]`) when passing array parameters as a general rule. Just pass them as pointers. In my experience it has never lead to confusion, and use of this convention is a standard practice.

Function Overloading

You can provide more than one function with the same name, as long as **the compiler has a way to figure out what function you are referencing *just by examining a function call*.** From this premise, you can figure out exactly what is legal and what is not in terms of overloads. You could memorize the rules, but there are subtleties that make for a good number of them. The best thing to do is really to just focus on that premise: you can then replace *knowing* with *understanding* which is always a good thing. As a simple self-test, suppose you have a function like `double avg(double x, double y)`. Which of the following would you suspect are legal overloads of `avg`?

1. `double avg(double x, double y, double z)`

[1]If you want to be really picky and technical, there is a subtle difference between `long[]` and `long*`: when you make a static array, like `long A[10];`, the pointer that corresponds to `A` *isn't actually stored anywhere.* Whenever you need to reference the pointer that would correspond to `A`, you get the address of `A[0]`, just as if `A` had datatype `long*`. But in my experience, this distinction doesn't have much practical consequence. Maybe it will make you sound smart in an interview though?

2. `double avg(double a, double b)`

3. `long avg(double x, double y)`

4. `double avg(long x, long y)`

Now suppose you have some function `bool fn1(long x, double y)`. Would
`bool fn1(double x, long y)`
be ok?

The Basics of Classes in C++

The real point behind classes in C++ is their role in abstract, object oriented programming (which we're not quite ready for at the moment) but at a minimum, classes allow you to *define your own datatypes* which is already very useful.

We're all very familiar with the intrinsic datatypes (`long`, `double`, `char`, etc.) but as we'll see, there is often a desire for more. Making our own datatypes greatly simplifies our code, letting us write programs that line-up a bit better with the abstract constructs in our minds. Strictly speaking, it isn't ever necessary, but it is often useful enough to seem like it. We could of course write all of our programs using nothing more than the intrinsic types (or better yet, just program in machine language)- but it doesn't mean that we should. A more refined, abstract approach is often a better idea. Typically one dives into the lower levels of abstraction only when additional performance is needed.

For example, suppose we want to do some operation on complex numbers. Let $x, c \in \mathbb{C}$ and consider a sequence of the form

$$(\cdots((x^2+c)^2+c)^2+c\cdots)^2+c$$

i.e., the sequence is defined by the recurrence $a_0 = x$ and $a_i = a_{i-1}^2 + c$. Suppose for a particular seed value x we'd like to see (via computational approximation) if this sequence diverges. We'll use the following method: compute up to n terms of the sequence, measuring the magnitude at each step. If the magnitude is larger than some threshold t, then we'll assume it diverges. What would the code for this look like?

```
bool willDiverge(double reX, double imX, double reC, double imC) {
    const double t = 8;
    unsigned long i;
    double reRslt,imRslt;
    for(i=0; i<n; i++)
    {
        // first, compute x^2
        reRslt = reX*reX - imX*imX;
        imRslt = 2*reX*imX;
        // now add c
        reRslt += reC;
        imRslt += imC;
        // compute the magnitude so we know if we should return
        if(reRslt*reRslt+imRslt*imRslt > t)
            return true;
        // finally, update with the new value for x:
        reX = reRslt;
        imX = imRslt;
    }
    return false; // made it through all the iterations
}
```

This code is manageable, but it isn't very pretty. Really, we're just doing basic arithmetic, so it seems like there should be an easier way. The problem is that there is no intrinsic datatype that represents $\mathbb{C}$. Fortunately, C++ gives us a way to make our own datatypes.

```
class complex {
public:
    //first we'll define the data members.
    //To store a complex number, we'll use two doubles:
    double re;
    double im;

    //now we'll define the public interface.
    //what do complex numbers do?
    complex(double r=0, double i=0); //constructor
    ~complex(); //destructor

    complex operator+(const complex& z);
    complex operator-(const complex& z);
    complex operator*(const complex& z);
    complex operator*(const double s); //function overloading
    complex operator/(const complex& z);
    double norm();
}
```

Now, using our fancy complex number datatype, we could have re-written the code as follows:

```
bool willDiverge(complex x, complex c) {
    const double t = 8;
    unsigned long i;
    for(i=0; i<n; i++) {
        // compute x^2 + c
        x = x*x+c;
        if(x.norm() > t)
            return true;
    }
    return false; // made it through all the iterations
}
```

Much nicer, right? Indeed. It gives a very pleasing semantic and syntactic representation. However, keep in mind that if your application needs to be extremely high-performance, this may not be the way to go.

Pointers

At this point, you should be familiar with quite a few C++ datatypes (e.g. `long, int, double, float, unsigned char...` and in the last section, we covered ways to make your own (via `structs` or `classes`). Often times it is convenient to explicitly keep track of where (in main memory) some variable of ours is being stored. How does one accomplish this? Well, every datatype has a corresponding pointer datatype for exactly this purpose. Pointers are variables that store memory addresses. *Think of them just as any other datatype.* There really is nothing special about them- they are not scary, they are not complicated, they are just variables that store memory addresses. Ok, let's get on to a few details. So, I mentioned that every datatype has a corresponding pointer datatype, but what is the name of it? Just append a star (`*`) to your existing datatype, and you have a pointer datatype. Easy, right? Here's an example:

```
int x; //x is an integer.
int* p; //p is a pointer to an integer.
```

Alright, so pointers store memory addresses. If I'm going to make much use out of that feature, there are two things I certainly need to be able to do:

1. Given a pointer (i.e., an address) I should be able to read the contents of memory at that address.
2. Given a variable, I should be able to obtain the address where it is stored.

C++ gives us the `*` operator (a.k.a. the "dereference" operator) and the `&` operator (a.k.a. the "address of" operator) for these purposes, respectively. They can be used as in the following example:

```
int x = 23; //x is an integer.
int* p; //p is a pointer to an integer.
p = &x; //p now holds the address of our variable x.
cout << x << "   " << *p << endl; //should print 23   23
```

For any variable, say `x`, the symbol `&x` will evaluate to the address of `x`. So, if `x` has datatype `int`, then `&x` has datatype `int*`. And for any pointer, say `p`, the symbol `*p` will evaluate to whatever is at the address stored in `p`. So if `p` has datatype `int*` then `*p` has datatype `int`. Pretty simple, right? You can think of the `*` and `&` operators as sort of inverses. For any datatype `K`, we have

$$\begin{tikzcd} \mathtt{K*} \arrow[r, shift left, "*"] & \mathtt{K} \arrow[l, shift left, "\&"] \end{tikzcd}$$

such that $* \circ \& = \mathbf{1}_{\mathtt{K}}$ and $\& \circ * = \mathbf{1}_{\mathtt{K*}}$. One more time, for emphasis, you should interpret these operators as follows:

$$\&\mathtt{x} \equiv \text{“tell me where } \mathtt{x} \text{ is in memory”}$$

$$*\mathtt{p} \equiv \text{“go get me what's at memory address } \mathtt{p}\text{”}$$

Self test: In the following code segments, what is the output?

```
int main() {
    long *p, x = 3;
    p = &x;
    cout << x << endl;
    cout << *p << endl;
    x = 4;
    cout << x << endl;
    cout << *p << endl;
    *p = 59;
    cout << x << endl;
    cout << *p << endl;
    return 0;
}

int main() {
    long *p1, *p2, x1 = 3, x2 = 2;
    p1 = &x1;
    p2 = &x2;
    cout << x1 << " " << *p1 << " "
         << x2 << " " << *p2 << endl;
    p1 = p2;
    cout << x1 << " " << *p1 << " "
         << x2 << " " << *p2 << endl;
    x2++;
    cout << x1 << " " << *p1 << " "
         << x2 << " " << *p2 << endl;
    p2 = &x1;
    x1 = 10;
    x2 = 23;
    cout << x1 << " " << *p1 << " "
         << x2 << " " << *p2 << endl;
    return 0;
}
```