SI 413, Unit 7: Functions

Daniel S. Roche (roche@usna.edu)

Fall 2018

This unit is all about function calls. Your work of implementing function calls in the labs should open your eyes to what's really going on with a function call, but there are even more options in how this could work than what we have been able to explore with SPL.

Functions fundamentally consist of two parts: the function *definition* and the *call site* (wherever the function is actually called). Of course there may be many call sites for a single function definition. Making a function call is like saying, "Go back and execute that code I already wrote down."

We are going to examine the different methods of communication between the call site and the function definition, which mostly means talking about how arguments work. Then we'll look at some more advanced uses of functions and other related concepts.

1 Parameter Passing Modes

Readings for this section: PLP, Section 8.3

It is possible for a function to just do the same thing every time, like print some fixed message to the screen. But this is unlikely. Usually, there is some *communication* between the call site and the function definition. For example, you might write a very simple function that just takes its argument (a number), multiplies it by two, and returns the result. In this scenario, the call site passes in the value of the original number, and the function passes back the value of the result, so there is *two-way communication* going on.

Function parameters and return values are the primary way in which a program communicates between the function *call site* and the function definition. The other option of course is to use commonly-accessible variables such as global variables: these are set before the function call to pass information into the function, and read after the function call to get information out.

Here is a simple C++ program that shows three ways of writing the "multiply by two" function and printing the result.

```
int foo1_global;
void foo1() {
  foo1_global = foo1_global * 2;
}

int foo2(int in) {
  return in * 2;
}

void foo3(int& inout) {
  inout *= 2;
}

int main() {
  int x=1, y=2, z=3;
```

```
    foo1_global = x;
    foo1();
    cout << foo1_global << endl;

    cout << foo2(y) << endl;

    foo3(z);
    cout << z << endl;

    return 0;
}
```

Notice the three basic ways of function communication:

- foo1 uses the global variable foo1_global to communicate in both directions. This idea will work with any kind of function, but it is usually not the best idea. The main reason is *scoping*. For instance, if there happens to be a local variable with the same name at the call site, we will get different behavior in lexical or dynamic scoping. Even more troubling, because this method is side-stepping the normal nice things that scoping gives us, recursion becomes impossible! So while using a common (global) variable for function communication might be useful sometimes, it's rarely the best option.
- foo2 does what we probably expect to see in C++: It takes the input as one argument, and returns the result separately. This is called "pass by value", and it means that the function gets a *copy* of the argument, so the argument is a one-way communication from call site to function body. Similarly, the return value is one-way communication back the other way. This works great in this situation, but can lead to difficulties when the arguments are large data structures (so we don't want to copy them), and when we want to return more than one value from a single function.
- foo3 uses one little & character to indicate that its argument is to be passed *by reference*. What this means is that the function parameter inout is actually referring to exactly the same thing as the variable z that gets passed in from main. In this way, the parameter is used for two-way communication between the call site and the function body. However, this option also has disadvantages: it constrains us to having only certain things (like variable names) as arguments, and it can lead to bugs as whoever called that function might not expect the argument to change.

The difference between foo2 and foo3 is called the "parameter passing mode", which means how the argument is passed from call site to function body.

The most typical option is *pass by value*; this is what happens by default in C++ and in our SPL language too: the function receives a copy of the argument. Any changes the function makes to the argument are not seen by the call site, because those changes are on a copy that only the function gets to use. This allows any arbitrary expression to be used as the argument, since it will simply be evaluated and then this value is bound to the function's parameter name. The advantages of pass by value are that it is relatively simple to implement, and it clearly defines the communication from call site to function body.

The second primary option for parameter passing is *pass by reference*, which is supported in C++ by using reference parameters and the & specifier. Some languages, including Perl and Visual Basic, actually use this mode by default. In pass by reference, the function parameter becomes an *alias* for the argument; any changes the function makes are immediately reflected at the call site. This overcomes some disadvantages of pass by value, namely by avoiding the need to copy large data structures as arguments, and by allowing a function to return more than one thing.

But pass by reference can also lead to programming errors, because it blurs the lines of communication between call site and function body. Is this argument for communicating from call site to function call, or from function call back to call site, or both? We simply can't know without looking at how the function actually works. Different languages provide some *compromises* for getting the nice aspects of pass by reference without all of the dangers:

- In C++, we can use the const keyword to specify that an argument will not be modified by the function, like cpp vector<string> rev(const vector<string> &A) { ... } This avoids the need to copy a large data structure, but also makes it clear that we still have one-way communication, since the function can't modify what's there.
- In Ada, the parameter passing mode for each argument is indicated as either in, out, or in out. Arguments specified as in are set by the call site and cannot be modified by the function; they represent one-way communication. Arguments specified as out are *not* set by the caller, but are set by the function; they represent one-way communication in the other direction. Arguments specified as in out are a little funny: they are not passed by reference, but in a different way called *pass by value/result*. What this means is: the argument is passed in as a copy (*by value*), but then when the function returns, whatever value is in that copy is *copied back* to the original parameter. This usually gives the same behavior as pass by reference, except that we still have to do the copying.
- In Java, primitive types like int are passed by value, just like in C++. But objects in Java are different; calling a function on a Java object is sort of like passing a pointer to that object. If the function modifies the object, then the original one gets modified too (like pass by reference). But if the function *reassigns* the argument to something else, the original isn't changed at all (this is like pass by value). What is going on?

  Well, you can read more details in your book or by looking at any Java reference, but this idea is called *pass by sharing*. The idea is that there is some data (in this case, the object itself) that is "shared" between the call site and the function body. It's sort of like pass-by-value, but with a "shallow copy" instead of a full copy. This can be really nice because it avoids the overhead of copying large data structures, while also giving some of the advantages of pass by value, but this "hybrid" approach to parameter passing is also notoriously confusing for Java programmers: see here for some evidence of the confusion.

In summary, you should know how pass by value, pass by reference, and pass by value/result work, and where we might want to use each one. You should also know that pass by sharing is another option that is used in some languages, and what pass by sharing means.

2 Parameter Evaluation

Readings for this section: PLP, Sections 6.6.2 and 10.4

Now we know how the value of an argument gets passed between the call site and the function body. But what about when the argument is an expression like (x + 2)? When exactly does this get evaluated? Of course there are multiple options! Here are the two basic choices:

- **Applicative order**. This is what you have come to expect from most programming languages you have used (C, Java, Scheme, SPL, ...). The argument expression is evaluated exactly once, just before the function is called. Then the resulting value is used in the function according to the parameter passing scheme (see above!).
- **Call by name**. This is the other extreme, and it exists in macro languages such as the C preprocessor (we'll discuss this below) and ALGOL. Instead of evaluating the argument at the beginning, that expression is only evaluated whenever the argument is *used*.

  The potential advantage here is that, if we *don't* use some particular argument, then we never have to waste the time evaluating it! The downside is basically the same thing: if you use an argument more than once, then you have to evaluate it more than once too.

The other option is called *normal order evaluation*, wherein the arguments are not evaluated until they are needed. A related concept (which for the purposes of this course will be *the same concept*) is called **lazy evaluation**, which means every argument is only evaluated at most once, but might not be evaluated at all if it's not needed. So lazy evaluation sort of gives the best of both worlds from above - unused arguments are never evaluated, but frequently-used arguments are only evaluated once!

So why not just use lazy evaluation everywhere? Why doesn't C++ use it? The issue comes when expressions have side-effects. For example, in C++ we can have a function call like foo(++x). The clear intent of the programmer in this case is that x should be incremented exactly once when this function call is made. But if we used lazy evaluation, x might not get incremented at all, depending on whether foo actually uses it! This is why lazy evaluation is supported by *pure functional languages* such as Haskell and ML, where referential transparency is strictly enforced. When there are no side effects, we can always do lazy evaluation. Hooray!

3 Overloading and Polymorphism

Readings for this section: PLP, Sections 3.5, 3.7, and 9.4

Function overloading and polymorphism are really two different things, but they are often confused by programmers. The common idea between them is having *different functions with the same name*. You should already be familiar with both of these concepts, but let's look at them from the standpoint of compiler writers.

3.1 Overloading functions

In function overloading, the same name is used for multiple functions, for example (in C++):

```
void bar (int x) { cout << "First version" << endl; }
double bar (double a, double b) { return a + b; }
int bar (string s) { return s.length(); }
```

Yes, that is a valid C++ program that compiles, no problem. We have three completely different functions, with different arguments and return types, but they are all called bar. So the question is, when I make a function call like bar(y), which version of the function gets called?

The way C++ (and Java) distinguish between the versions is by looking at the type and number of arguments. So if we do bar(5), it will call the first version, bar(1.2, 3.4) will call the second version, and so on. This should also tell you why there is no function overloading in languages like Scheme - without any declared types, there's no way of telling which version you would want to call!

C++ also has a special feature (not shared by Java) that the built-in operators like < and + can be overloaded for newly created types. For example:

```
struct Point {
  int x;
  int y;
};

// overload the + operator so we can add Point objects together
Point operator+ (Point a, Point b) {
  Point result;
  result.x = a.x + b.x;
  result.y = a.y + b.y;
  return result;
}

int main() {
  Point p1, p2;
  /* ... */
  Point p3 = p1 + p2; // calls our version of the + operator.
  int x = 1 + 2; // and of course this will still work as usual
}
```

This is a special quirk of the C++ language, but it's important to recognize what's special and what's not. The special thing is that all the operators in C++ are really just function calls with special names like operator+. The function overloading mechanism is just the ordinary one, on these very special kinds of functions.

3.2 Overloading methods

Overloading works with class methods in mostly the same way as any other function, but it looks differently because class methods are called differently than other functions.

Say we have two Java classes called Mid and Prof, and they both have a method void pay(double amt); that produces a paycheck in the indicated amount. These classes might be used like:

```
Mid firstie = new Mid();
Prof roche = new Prof();
Prof aviv = new Prof();
firstie.pay(.05);
if (! furlough) {
  roche.pay(1000000.00);
  aviv.pay(-10.00);
}
```

We have these calls to a function called pay, and they each take one argument (a double). But since the arguments have the same types, how can overloading be used to distinguish these functions?

The answer is that we are ignoring the "hidden" first argument to these functions - the object itself! A function call like firstie .pay(.05) is essentially calling a function called pay with two arguments: firstie and .05. The types of both of these arguments (Mid and double, respectively) are used by the compiler to determine which overloaded function will be executed.

3.3 Polymorphism

Polymorphism is one of the most important aspects for object-oriented programming, and it is sort of the intersection of overloading and inheritance. You should already be familiar with the idea, but now we will understand a little bit better how it actually works.

Let's go back to the previous example, and suppose that both classes Mid and Prof are subclasses of a common base class:

```
abstract class DoD {
  public abstract void pay(double amt);
}
```

Then we could treat all of those like DoD objects and do something like:

```
DoD[] employees = new Dod[] { new Mid(), new Prof() };
for (DoD emp : employees) {
  emp.pay(100.00);
}
```

So the question again is which pay method will be called here. Unlike before, the *declared* types of the arguments cannot be used to determine which function, because the type of emp is always DoD and the type of 100.00 is always double.

This is where *subtype polymorphism* comes to the rescue, and uses the principles of inheritance to determine which method gets called. In this case, the *actual type* of each DoD object is looked up to determine which method to call. Since we're talking about the actual value of these variables, not just the declared type, *this look-up can only be done at run time.*

The specific way this gets implemented in C++ and similar languages is through the use of something called a *virtual methods table* or vtable. You can read about the details in your book (or ask your instructor!), but the general idea is that each object stores in memory (at run-time) pointers to the actual methods that should be called, depending on the type. So the vtable of firstie will point to the pay() method in Mid, whereas the vtable for roche will point to the method in Prof instead. This causes a small performance penalty since the functions must be looked up at run-time, rather than being hard-coded in at compile time.

4 Operators, Built-ins

We just discussed how operator overloading in C++ is just like any other function overloading because, in that language, operators are just a special kind of function call.

You have actually used this feature quite a bit if you've done any C++ programming with input/output streams like cin and cout. You know that you can do something like

cout << 1 << " is the loneliest number.\n";

What's really happening here? First, you have to know that << is an operator in C++. By default, it's the left-shift operator for doing bitwise shift operations: 5 << 3 is equal to 101 (binary 5) with three more zeros, so 101000, which is 40.

But obviously there's no bit shifting going on in the example above. This is because the operator has been *overloaded* in the C++ iostream library with these two functions (and many more):

```
ostream& operator<< (ostream& out, int x); // prints out an int
ostream& operator<< (ostream& out, const char* s); // prints out a C-string
```

Basically, if the left-hand side argument of a bit-shift operation happens to be of type ostream (by the way, cout is an ostream), then those overloaded functions get called to do output instead of bit-shifting.

Besides this particular application, more generally operator overloading in C++ lets you do some really cool things with user-defined types, and essentially treat them just like the built-in types such as int.

However, this is a pretty special property of C++, and operators can't be overloaded like this in most other languages. In Java, you can't overload operators because operators are not ordinary methods - so there's no function to overload. In Scheme, we have the opposite barrier: even though operators such as + and < are ordinary functions (which can be redefined), because there are no declared types we cannot overload these functions with multiple versions. That is, in Scheme, you can redefine + in some new way, but you can't just tell Scheme how to do + on some new type.

And this brings us to the topic of **built-in functions**. Obviously we know that most languages let you write your own functions and then call them. These are called user-defined functions. There are usually also other methods that you might be able to call after some kind of include directive, like printf in C or java.util.Arrays.sort(A). These are functions that are programmed in the same language, just not by you, and they are called *library routines*.

But in most languages there is another class of methods or functions that are not written in that language at all. Rather, these are hard-coded into the interpreter or compiler itself. Consider, for example, the addition of two integers. This looks like any other operator, but it is clearly a low-level function that at some level needs to be translated to some sort of ADD instruction on the processor.

Such functions that come included in the language *and* are not written as library routines in the language itself are called "built-in functions", since they are actually built in to the compiler or the interpreter. Usually the low-level system calls and such are written this way, since it would be impossible to express them in the language itself.

(The exception to that rule is in C/C++ in a Linux environment. Since the Linux operating system kernel itself is written in C, even low-level system calls are essentially library routines in this case.)

5 Macros

Readings for this section: PLP, Section 3.7

Earlier we mentioned that "call by name'' is an alternative to applicative and normal order evaluation strategies, whereby the actual expression gets re-evaluated whenever that variable is used, possibly multiple times. This is actually what happens with *preprocessor macros* in a language like C++. Consider the following example:

```
int y = 10;

#define X (y + 2)

void foo(int y) {
  cout << X << endl;
}

int main() {
  cout << X << endl; // prints 12, OK
  y = y * 20;
  cout << X << endl; // prints 202, hmmm
  foo(50); // uses foo's local variable y, prints 52
}
```

You can actually compile and run that code and see that it really does behave as described in the comments. So clearly X is not just any other variable, or else it would be assigned the value 12 at the beginning and print that out every time.

What's going on has to do with the C++ preprocessor, which is really a fast but "dumb" pre-compilation step that carries out all the commands that start with a # sign. You are very familiar with #include directives, which tell the preprocessor to find some file and dump its contents out into the current source code.

The #define directive is used to declare a *macro*. Since the C preprocessor is dumb, it doesn't know anything about the C++ language, types, variables, anything like that. All it does is replace all tokens that match the name of the macro (in this case, X) with the value of the macro (in this case, (y + 2)). So the above program will be translated by the preprocessor into the following *before* it actually gets compiled:

```
int y = 10;

void foo(int y) {
  cout << (y + 2) << endl;
}

int main() {
  cout << (y + 2) << endl;
  y = y * 20;
  cout << (y + 2) << endl;
  foo(50);
}
```

Now it makes sense! Macros can be very useful especially for constants like PI because you know that number will be "hard-coded" into your program, but you don't have to type it out 10 different times. But there are also potential pitfalls. For example, if we define a macro like

```
#define X 1+2
```

and then use it like

```
cout << 3*X << endl;
```

this will not print out 9 like you might expect, but rather 5. Why? Because the 3∗X is textually expanded by the preprocessor to 3∗1+2, which the compiler then uses operator precedence on to compute (3 * 1) + 2, which is of course 5. This is why it's usually a good idea to put your constant macros inside a pair of parentheses, although even that might have some unforeseen consequences.

There is even another level to this insanity in what are called *function-like macros*. These are also translated by the preprocessor, textually, before compilation, but the difference is that they take arguments! There are some examples of these on the slides from this unit.

The important thing to remember is that all these macro tools are just doing textual substitution without any knowledge of C++ *semantics*, and it all happens before the compilation step. This means tremendous opportunities for efficiency gains, but also tremendous pitfalls and confusing error messages.