Creating Functions in C/C++

Motivation

There are many reasons to create functions for your programs.

- A fragment of code that appears in your program in multiple places can be placed into a function definition and replaced by a function call instead, resulting in a smaller, more maintainable program. It is smaller because there are fewer lines of code. It is more maintainable because if you decide to change the code, you only have to do it in one place, instead of searching through the entire program for all occurrences of that code fragment. This makes it less error-prone, since it is possible to miss one of the fragments if they are dispersed throughout the program.
- By creating a function and putting a code fragment into it, the program becomes easier to read and more modular. For example, if you have code that displays a menu on the screen and gets the user's entered choice, you could create a function named `GetUserSelection()` that returns the user's choice, making it obvious to the reader what the code does. The program becomes more modular because it now contains another separately maintainable function.
- Functions can be reused more easily in other programs. You may find that you need the same code fragment in many programs. By creating a function that encapsulates it, you can put that code into other programs without having to rename variables and reorganize the program code.

There is one downside to creating a function: the ***function call overhead***. I will discuss this later.

How do you create a function?

To define a function in your C or C++ program, you write a ***function definition***, which has the form

```
result_type function_name ( parameter_list)
{ function_body }
```

where

- *result_type* is any type such as `int`, `double`, `char`, or `string`, but may also be the word `void`.
  If the result_type is `void`, it means the function does not "return" anything.
- *function_name* is any valid C++ identifier
- *parameter_list* is a list of the form *typespec parameter, typespec parameter,... typespec parameter*
  where *typespec* is a specification of a type (such as `int` or `char` but might be a bit more complex than this) and *parameter* is in the simplest case just a C/C++ identifier. (In C++ there are things that can appear here that cannot appear in a C function.)
- *function_body* is just like the body of a main program. In fact `main()` is just a special function. It consists of declarations and statements.

The first line, which contains the result type, the function name, and the parameter list is called the ***function header***.

Examples

```
double eval ( double a, double b, double c, double x)   // The header
{   // The body
    // returns value of polynomial a*x^2 + b*x + c
    return a*x*x + b*x + c;
}

double volumeOfSphere ( double radius)                  // The header
{
    // returns the volume of a sphere with given radius
    return 4*3.141592*radius/3;
}

void insertNewLines ( int N )                           // The header
{
    // insert N newlines into cout
    for ( int i = 0; i < N; i++ )
    std::cout << std::endl;
}
```

The first two examples are of functions that return something. This means that when they finish running, the value of the expression in the return statement is the value that they "return". The third example has a void return type. This means it does not return a value. It runs and does something, and it can even have return statements, but they cannot return a value.

Where do you put function definitions?

In the beginning, you should put all function definitions *after* all `#include` and `using` directives but *before* the `main()` program. Once your understanding is solidified, you will put them *after* `main()` but will put a ***function prototype*** before `main()`.

How do you use functions that you define?

You use them the same way that you use library functions, by putting *calls* to them in the code. The program that contains the call is the ***caller***, and we say that it ***calls*** the function. For example the following partly incomplete program calls the first function.

```
int main()
{
    double A, B, C, x;
    // get values A, B, C, and x here
    cout << "The value of the polynomial is " << eval(A, B, C, x)  << endl;
}
```

In the function call, you must make sure to put the arguments in the correct order. Parameters are positional, which means that it is their position in the list that matters, not their names. In the above call, the value of `A` is copied into the parameter `a`, `B` into `b`, `C` into `c`, and `x` into `x`, and then the function executes. When it returns, the value that it returns is used in the caller wherever the call was written. For example, if `A` = 1, `B` = 8, `C` = 16, and `x` = -4, then `eval(A,B,C,x)` returns 0, and it would be the same effect as if the program had the line

```
cout << "The value of the polynomial is " << 0 << endl;
```

If you change the order of the arguments, the result will be different. If you call `eval(C,B,A,x)` you will get a different answer.

Some Guidelines About Writing Functions

- Functions that return values should not have side effects. A side effect is a change to the values of variables in the program other than those that are declared within the function or in its parameter list.
- Functions should do one thing only.
- Functions should do one thing completely.
- Functions should have meaningful names.
- Functions should be grouped together in the program file.
- Functions should always have comments that summarize its purpose, pre-conditions and post-conditions, and what it returns.
- If a function is more than 50 to 100 lines, it is probably too long and should be broken into smaller functions.
- Never write a function that is essentially a main program just to call it from `main()`.

Examples of Functions Not to Write

```
double workerPay( double salary)
{
    cout << "The pay is $";
    return salary;
}
```

This just prints something that could be printed by the caller and it returns what it was passed. It has a side effect (it changes `cout`) and it returns a value too.

```
void mysqrt( double num )
{
    cout << "The square root of " << num << " is " << sqrt(num) << endl;
}
```

This just calls a library function and prints some stuff. No need to do this.

Can functions call other functions?

They certainly can, and often do. In fact `main()` is a function and `main()` calls the functions you write as well as the ones in the libraries. But to give you a better idea, the following is perfectly legitimate, albeit silly, code.

```
void func1 ( int n )
{
    cout << "In func1: " << n << "\n";
}

void func2 ( int m )
{
    cout << "In func2: " << m << "\n";
    func1 (m);
}
```

```
void func3 ( int k )
{
    cout << "In func3: " << k << "\n";
    func2(k);
 }

void func4 ( int m, int n )
{
    func3(m);
    func2(n);
}
```

Can a function call itself?

Yes, but for now we will avoid this topic. It is perilous ground and we will visit it after you can walk, run, and jump on solid ground.

Scope

The variables (and other identifiers) in a program have various properties. You know about some of them already. For example, variables have ***associated type***, and at any given instant of time, they have a ***value***. They also have ***storage requirement***s, i.e., do they need two bytes, four bytes, or something larger? Another property associated with variables and program identifiers in general is their ***scope***.

The ***scope*** of a name is the part of the program in which the name can be used. You already know one ***scope rule***: if you use a for-loop such as this:

```
for ( int i = 0; i < 10; i++) {
    // do something here
}
```

you should know that the name `i` extends only to the end of the for-loop statement itself, i.e., to everything within the body of the loop, and no further. You should also know that this is C++ and not C. The C standard does not let you declare the index within the loop, although some compilers allow it.)

For now, you should know about two types of scope.

Block Scope: A block is the code between a pair of matching curly braces. An identifier declared inside a block has block scope. It is visible from the point at which it is declared until the innermost right curly brace containing the declaration. Function parameters have block scope – they are visible only within the body of the function.

File Scope: An identifier declared outside of any function including main is visible from the point of the declaration to the end of the file in which it was declared.

Local Variables

Functions can have variable declarations. The variables declared within functions have block scope – they are visible until the end of the innermost right curly brace, which is in the simplest case, the function body. For example, in each of the following functions

```
long arithmeticsum( int num)
{
    long sum = 0;
    int  j;

    for ( j = 1; j <= num; j++)
        sum = sum + j;
    return sum;
}

long sumsquares( int num)
{
    long sum = 0;
    int  j;

    for ( j = 1; j <= num; j++)
        sum = sum + j*j;
    return sum;
}
```

both `sum` and `j` have scope that extends to the end of the function block. They are called ***local variables*** and are said to have local scope. Each function has a variable named `sum` (and a variable named `j`). They are different variables that just happen to have the same name. The variables declared in your main program have local scope too. They are visible only from the point at which they are defined until the end of the curly brace that ends the main program block.

Global Variables

Variables declared outside of any function (which therefore have file scope) are called ***global variable***s. They are visible to all functions in the file from the point of their declarations forward. There are many reasons not to use global variables in programs, because it makes program harder to read, understand, debug, and maintain. The one exception to this is constant global variables. It is acceptable to define global constants in a program, if these definitions are placed at the very top of the file after the `#include` and `using` directives. This is because it makes them easy to see and makes changing them easier.

Example

```
// various constants used in the program
const double PI       = 3.14159236;
const int    MAXSIZE = 1000;
const string MESSAGE = "The file could not be opened.\n";

int main()
{
    // stuff here
    return 0;
}
```

Function Prototypes (Declarations): Moving Towards Data Abstraction

If a program has several functions in it and their definitions precede the main program, the main program ends up at the bottom of the file. This is inconvenient, because when you read a program you usually want to first read the main program to get a general sense of how it works. The functions in it are often solving

small problems while the main program provides structure. The C and C++ languages provide the means to declare the functions in the beginning and put the definitions of them elsewhere.

What do we mean by declaring and defining functions, and how are they different?

A function definition is what you just learned how to write. It is the function's header together with its block. The function header by itself does not say what the function does, but it provides enough information to the compiler so that it can check whether the calls to the function are valid. The function header, when it is terminated by a semicolon, is called a function declaration or function prototype.

How does having the function prototype help?

This is the nice part. We put all of the function prototypes at the beginning of the file, before main(), and the definitions after main(). The comments that describe what the function does stay with the prototype at the beginning of the file. They do not need to be repeated with the function definitions. The comments and the prototype ar all that a programmer needs to figure out how to call the function and what it does. The actual function body should be thought of as a black box, a hidden piece of code that does what the comments say it does, but which the programmer does not need to see to use the function. This is the essence of procedural abstraction, an important principle of good software engineering practice, and the underpinning of object-oriented programming.

Example

```
#include <string>
#include <iostream>
using namespace std;

/*************************************************************************/
/*                          Function Prototypes                          */
/*************************************************************************/

/* eval(a,b,c,x) returns value of polynomial a*x^2 + b*x + c      */
double  eval           ( double a, double b, double c, double x);

/* volumeOfSphere(r) returns the volume of a sphere with radius r */
double  volumeOfSphere  ( double radius);

/* insertNewLines(N)  inserts N newline characters into cout      */
void    insertNewLines   ( int N );


/*************************************************************************/
/*                             Main Program                              */
/*************************************************************************/

int main()
{
    // tbe main program's body is here
}

/*************************************************************************/
/*                          Function Definition                          */
```

```
/*************************************************************************/

double eval ( double a, double b, double c, double x)
{
    return a*x*x + b*x + c;
}

double volumeOfSphere ( double radius)
{
    return 4*3.141592*radius/3;
}

void insertNewLines ( int N )
{
    for ( int i = 0; i < N; i++ )
    std::cout << std::endl;
}
```

The next step after this is to put the function declarations into one file, called a ***header file***, and the function definitions into a second file, called the ***implementation file***. This will be discussed in a future lesson.

Call-by-Reference Parameters

What if we want to write a function that actually changes the value of its parameters? For example, suppose we want a function that can be called like this:

```
swap(x,y)
```

that will swap the values of arguments x and y. In other words if `x = 10` and `y = 20` before the call, then after the call `x = 10` and `y = 10`. If we write the function like this:

```
void swap ( int x, int y )
{
    int temp = x;
    x = y;
    y = temp;
}
```

will this do the trick?

Try it and you will see that it does not. Remember that when a function is called, ***the values of the arguments are copied into the storage cells of the parameters***. The function runs and when it terminates, the parameter storage cell contents are not copied back to the arguments. When you think about it, it would make no sense. Suppose we define a function `double()` like this:

```
int double ( int x )
{

    x = 2*x;
    return x;
}
```

and we call it with the call

```
cout << double(10);
```

If the value of parameter x were copied back to its argument, it would mean we could replace a constant literal 10 by the value 20, which is impossible.

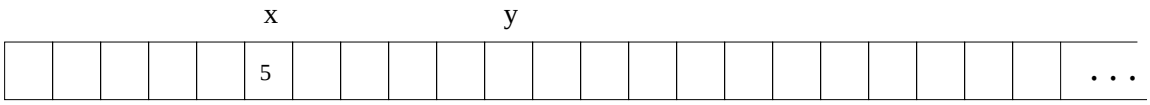
The kind of parameters we have been using so far are known as ***call-by-value parameters***. This is because the value of the argument is passed to them. So what is the alternative?

The Concept Of A Reference

We mentioned before that variables have several different properties, such as their type, their storage requirements, and their scope. Every variable also has ***contents*** and ***location***. In

```
int x = 5;
int y;
```

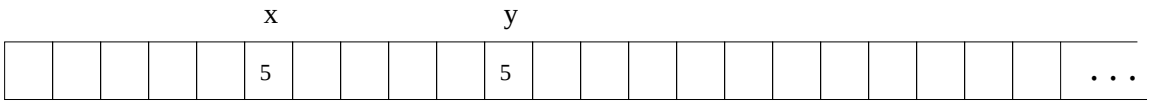
`x` is the name of a variable of type `int` with contents of `5`. `x` also has a location. We don't know its actual location exactly but we know that it has some storage location in memory and that location has a specific, numbered address. In the picture below, think of each box as a storage location capable of storing an integer:



The boxes, which are actually ***memory words*** (4-byte units), have addresses but we don't care what they are. The assignment statement

```
y = x;
```

causes the ***contents*** of `x` to be copied into the ***location*** of `y`[1]:



A reference variable is like another name for a location that already exists. A reference variable really stores an address. If we define two variables x and y as follows:

```
int  x = 5;
int& y = x;
```

then `y` is called a reference to `x`. The variable `y` stores the location, or address, of `x`, but it can be used in place of `x`. The two statements

```
cout << x;
cout << y;
```

[1] Notice the asymmetry of the = operator: the value of its right hand side operand is the ***contents of the operand***, but the value of the left hand side operand is the ***location of the operand***. In other words, putting a variable on the right hand side of "=" causes its value to be extracted whereas putting it on the left hand side causes its location to be found.

The ***lvalue*** of a variable is its location. The ***rvalue*** is its contents. lvalue and rvalue are just abstractions; they are not really stored anywhere, but they help to make things clearer.

print the same value because `y` is just a pseudonym, another name, for `x`. If we increment `x` and output the value of `y`, we will see it has changed as well:

```
x++;
cout << y;
```

and we can increment `y` and output the value of `x` and see that it has also changed.

```
y++;
cout << x;
```

The &-operator used in this way creates a ***reference variable***:

```
type & identifier = variable;
```

makes the identifier a reference to the variable. The reference variable type must be the same as type of the variable whose address is being assigned to it.

```
char c;
char & refTochar = c;
int x;
int &  refToint  = x;
```

are two valid reference declarations, but not this:

```
char c;
int & cref = c;
```

because `cref` is of type `int&` and it should be of type `char&`. It does not matter whether there is space to the left or right of the &-operator. The following three statements are equivalent:

```
int&  y = x;
int & y = x;
int  &y = x;
```

We can use reference variables as parameters of functions. They are then known as ***call-by-reference parameters***.

Examples

```
void swap( int & x, int & y)
// replaces x by y and y by x
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
```

```
    int a = 10;
    int b = 20;
    swap(a,b);
    cout << a << " " << b << endl;
    return 0;
}
```

This program will print 20 and then 10 because `swap(a,b)` caused the values of `a` and `b` to be swapped. This is because `x` is just another name for `a` in `swap()` and `y` is another name for `b`. Each assignment statement in `swap()` is actually altering the values of `a` and/or `b`.

```
void castActors( string & Romeo,
                 string & Juliet)
{
    Romeo = "Leonardo DiCaprio";
    Juliet = "Claire Danes";
}

int main()
{
    string lead_male_role;
    string lead_female_role;

    castActors( lead_male_role, lead_female_role );
    cout << "Romeo will be played by  " << lead_male_role << "\n";
    cout << "Juliet will be played by " << lead_female_role << "\n";
    return 0;
}
```

In this second example, the strings in the main program have no initial value but after the call to `castActors()`, they are given values. Call-by-reference parameters are the key to writing functions that must give values to multiple variables, such as functions that initialize many variables.

Overloading Functions

Functions can be overloaded in C++, but not in C. Simply put, it means that the same name can be used for two different functions, provided that the compiler can distinguish which function is being called when it tries to compile the code. This can be convenient sometimes, but most of the time there is little need for this feature of C++. Nonetheless, because you may be called upon to read a program that contains overloaded functions, I discuss them very briefly here.

The rule sounds simple at first: The same name can be used for two different functions provided that they have a different number of formal parameters, or they have one or more formal parameters of different types. So these are valid overloads:

```
int max( int a, int b );
int max( int x, int y, int z );
```

and so are these:

```
void sort( int & x, int & y );
void sort( double & x, double & y );
```

because the first pair have a different number of parameters and the second pair have different types for their parameters.

These are not valid overloads:

```
int max( int a, int b );
long max( int x, int y );
```

because they only differ in their return type. (The names of the parameters are irrelevant.) These are not valid either:

```
void sort( int & x, int & y );
void sort( int x, int y );
```

because the types are the same and kind of parameter passing is not used to distinguish them.

Overloading can get complicated because of type casting and type conversion. If you have two functions such as

```
void foo( int x, double y);
void foo( double x, int y);
```

and your program makes the call

```
foo(1,2);
```

which one should the compiler use? Guess what? It can't really decide either, so it generates an error. Unless you have a good reason to overload function names, it is best to avoid it.

Default Arguments

C++, and not C, lets you assign default values to the ***call-by-value parameters*** of a function, in right to left order. In other words, you can declare a function so that the rightmost parameters have default values, as in the following example:

```
string repeatedstr( string str, int numcopies = 1);
```

The calling program can omit the second argument, in which case `numcopies` will be assigned 1:

```
cout << repeatedstr("*");
```

will print a single '*' whereas

```
cout << repeatedstr("*", 10);
```

will print ten of them.

Note: You can not assign default values to call-by-reference parameters.

You can have any number of default arguments but the rule is that a parameter can only have a default argument if the parameter to its right has one. So these are valid:

```
void foo( int a, int b, int c = 1, int d = 2, int e = 3);
void bar(int a, int b, int c = 1);
```

but not these

```
void foo( int a = 1, int b);
void bar(int a = 1, int b = 2, int c);
```

Moreover, you can run into problems with overloading and default arguments, as in the following:

```
void f( int x, int y, int z = 1);
void f( int x, int y);
```

If the program calls `f(1,2)`, is it the first or the second function that will be invoked? Guess what? The compiler can't know either, so it generates an error.

The C++ `iostream` library has two prototypes for `getline()`:

```
istream& getline ( istream& is, string& str, char delim );
istream& getline ( istream& is, string& str );
```

Do you think it is overloaded, or do you think that `getline()` is really defined with the header

```
istream& getline ( istream& is, string& str, char delim = '\n' );
```

What do you think is the better solution?

The real benefit of being able to assign default values to parameters comes with class constructors, a topic that will be covered when we get to classes. My advice is that you avoid overloading unless you have a really good reason to use it.