

---

# C++ Programming Fundamentals

---

Elvis C. Foster

---

---

## Lecture 04: Functions

---

In your introduction to computer science, the importance of algorithms was no doubt emphasized. Recall that an important component of an algorithm is its subroutines (also called subprograms).

Subroutines give the software developer the power and flexibility of breaking down a complex problem into simpler manageable, understandable components. In C++, subroutines are implemented as functions.

This lecture discusses C++ functions. The lecture proceeds under the following subheadings:

- Function Definition
- Calling the Function
- Some Special Functions
- Command Line Arguments to main ( )
- Recursion
- Function Overloading
- Default Function Arguments
- Summary and Concluding Remarks

## 4.1 Function Definition

A function is a segment of the program that performs a specific set of related activities. It may or may not return a value to the calling statement. The syntax for function definition is provided in figure 4-1:

Figure 4-1: Syntax for Function Definition

```

FunctionDef ::=
<ReturnType> <FunctionName> ([<ParameterDef>] [*; <ParameterDef> *])
{
    <FunctionBody>
}

ParameterDef ::=
<Type> <Parameter> [* , <Parameter> *]

FunctionBody ::=
[<VariableDeclaration> [*<VariableDeclaration> *]]
<Statement> [* ; <Statement> *]
  
```

Essentially, the C++ function must have a return type, a name, and a set of parameters. The function body consists of any combination of variable declarations and statements. We have already covered some C++ statements, and will cover several others as we progress through the course. Following are some additional points of clarification:

1. The return type is any valid C++ type (primitive or programmer defined). The function will return a value of that type. If you do not wish to have the function return a value, then the return type must be specified as **void**.
2. Formal parameter(s) is (are) variables that the function will treat as input to it. The function will use the value(s) in this/these parameter(s) to perform its operations. Parameters are declared in manner that is similar to variable declaration. Actual parameters (arguments) corresponding to the formal parameters must be supplied when the function is called. On the function call, each argument is copied to the corresponding parameter.
3. The body of the function may contain any combination of valid C++ declarations and statements. If the function is expected to return a value to the calling statement, then at least one statement must be a **return-statement**. The **return-statement** has the following syntax:

```

ReturnStatement ::= return <Expression>;
  
```

4. C++ does not allow a function to be defined within another function; all functions are autonomous.

## 4.1 Function Definition (continued)

5. A function prototype is a declarative statement that alerts the C++ compiler of function to be subsequently defined and called in that program file. The prototype is simply the function heading, terminated by a semicolon. The defined function must match the previously declared function prototype. It is not necessary to specify parameter names in the function prototype, but this is normally done in the interest of clarity. Function prototypes are usually declared ahead of the **main()** function.

**Example 4-1:** The following code includes two functions — **main()** and **power(...)**

```
#include <iostream.h>

double power (int m, n); // Function prototype
int base, exp; double result;
...
void main()
{
    ...
    cout << "Enter the Base, followed by the Exponent:";
    cin >> base; gets();
    cout << "Enter the the Exponent:";
    cin >> exp; gets();

    // Output the calculation result
    cout << base << " raised to the power of " << exp << " is: ";
    printf ("%d %d %d", base, exp, power (base, exp));

    /* is equivalent to
    result = power (base, exp);
    cout<< base << exp << result; */
    ...
}

// Function definition for Power
double power (int b, e)
{
    int i; double p;
    p = 1;
    for (i=1; i <= e; i++) p = p*b;
    return p;
}
```

## 4.2 Calling the Function

---

If the function returns a value, it can be called by its name and any required argument(s) in any valid (arithmetic or Boolean) expression or assignment statement. If the function does not return a value, it can be called by simply specifying its name and any required argument(s).

Remember, function arguments are always specified within parentheses. Moreover, function arguments must be specified in an order that corresponds to the function parameters, as specified in the function definition and function prototype. If the function requires no argument, then an empty pair of parentheses must be specified with the function name.

Here are a few things you need to remember when working with C++ functions:

1. Typically, arguments are passed “by value” only. This means that argument values are stored in temporary variables (parameters), confined to the particular function rather than the original variables. The called function therefore cannot directly alter an argument being passed to it; it can only alter a private temporary copy. On the function call, the arguments are copied to the corresponding parameters.
2. Variables defined within any given function are known only to that function; they cannot be referenced by any other function. This applies to the function **main( )** also.
3. There are several ways to get around these constraints. Following are five possibilities:
  - a. Call the function via an assignment statement.

**Example:**

```
thisValue = thisFunction (Arg1, Arg2);
```

- b. Declare the variables globally, so that they are accessible from any function (this strategy should not be used indiscriminately, but only when it can be justified).
  - c. Pass pointers to the variables to be changed as arguments to the function. Within the function, the pointer(s) is (are) manipulated as required.
  - d. Use *reference parameters* in the function; the reference parameter creates an implicit link to the arguments supplied to the function when it is called.
  - e. The function could call a member function of a class, where this member function has the responsibility of changing the current instance of the class. This will become clear when we discuss classes (lecture 6).
4. Rather than passing pointer(s) to a function, a more elegant approach is simply to declare the parameter as a reference parameter in the function definition (and prototype). This is done by putting an ampersand (&) in front of the parameter. Within the function, operations performed on the reference parameter affect the argument used to call the function, not the reference parameter itself. [There is an implicit link between the argument and the reference parameter]. In that sense, the reference parameter is simply an alias of the actual argument used to call the function.

## 4.2 Calling the Function (continued)

**Example 4-2:** The following function updates a pointer to an integer.

```
void smaller (int thisVal, thatVal; int * smallerI)
{
    if (thisVal < thatVal)
        smallerI = &thisVal;           // *smallerI = thisVal
    else smallerI = &thatVal;           // *smallerI = thatVal
}

/* This function updates a pointer which points to the address of the variable that is smaller.
To call this function, specify the arguments with an ampersand (&) on the pointer arguments.
e.g. smaller (this, that, &smallerThisThat); */
```

**Example 4-3:** The following function returns the smaller of two integer values received as parameters.

```
int smaller (int thisVal, thatVal)
{
    if (thisVal < thatVal) return thisVal;
    else return thatVal;
}

/* This function returns the smaller of two values received. It must be called by including it in
an expression or assignment statement*/
```

**Example 4-4:** The following function updates a global variable.

```
int smallerI;
// ...
void main ( )
{
    int number1, number2;
    // ...
    Smaller (number1, number2);
    // ...
}

void smaller (int thisVal, thatVal)
{
    if (thisVal < thatVal) smallerI = thisVal;
    else smallerI = thatVal;
}

/* This function updates a global variable. It must be called by simply specifying its name
with the appropriate arguments e.g. smaller(this, that); */
```

## 4.2 Calling the Function (continued)

**Example 4-5:** The following function (indirectly) updates the referenced parameter called **smallerI**.

```
void smaller (int thisVal, thatVal, int &smallerI)
{
    if (thisVal < thatVal) smallerI = thisVal;
    else smallerI = thatVal;
}

/* Note, the operator & is not required within the function, once the parameter is declared as a
referenced parameter */

/* To call the function, simply issue the function name and arguments (without the & sign) e.g.
smaller (this, that, smallerI); */
```

## 4.3 Some Special Functions

We have already introduced several library functions and many more will be introduced as we proceed. All the header files (review section 1.4) contain functions which are applicable to various situations. Below are some commonly used functions from various header files (some of these functions have already been discussed; others will be discussed later in the course):

**Figure 4-2: Commonly Used Standard Functions**

Header File <math.h>	
Function	Comment
double cos (x)	Returns the cosine of x
double sin (x)	Returns the cosine of x
double tan (x)	Returns the tangent of x
double cosh (x)	Returns the cosh of x (hyperbolic cosine function)
double sinh (x)	Returns the sinh of x (hyperbolic sine function)
double tanh (x)	Returns the tanh of x (hyperbolic tangent function)
double asin (x)	Returns $\sin^{-1}(x)$
double atan (x)	Returns $\tan^{-1}(x)$
double atanz (y, x)	Returns $\tan^{-1}(y/x)$
double log (x)	Returns $\ln(x)$
double log10 (x)	Returns $\log_{10}(x)$
double exp (x)	Returns $e^x$
double pow (x,y)	Returns $x^y$
double sqrt (x)	Returns square root of x
double fabs (x)	Returns absolute value of the floating point x
int abs (x)	Returns absolute value of the integer x
long labs (x)	Returns absolute value of the long integer x

Figure 4-2: Commonly Used Standard Functions (continued)

Header File <stdio.h>	
Function	Comment
printf, scanf, sscanf, gets, getchar, putchar	Review section 1.7
fopen (Filename, Mode)	Returns the file pointer
fclose(Filepointer)	Closes the file
fprintf(Filepointer, Format, Argument)	Writes formatted data to a file
fscanf(Filepointer, Format, Argument)	Reads formatted data from a file
fgetc(Filepointer)	Reads a character from a file stream
fgets(Filepointer)	Reads a string from a file stream
fputc(aCharacter, Filerpointer)	Writes a character to a file stream
fputs(aString, Filerpointer)	Writes a string to a file stream
int fseek(Filepointer, Offset, StartPosition)	Positions the file pointer
int fsetpos(Filepointer, PositionPointer)	Positions the file pointer
int fgetpos(Filepointer, PositionPointer)	Returns positions the file pointer
int fread(DataPointer, Itemsize, NumberOfItems, FilePointer)	Reads block data from a file
int fwrite(DataPointer, Itemsize, NumberOfItems, FilePointer)	Writes block data to a file
int rename(const char* oldN, const char* newN)	Renames file <b>oldN</b> to <b>newN</b>
int remove ( const char* Filename );	Removes the specified file; returns 0 if successful; or <0
Header File <iostream.h>	
Function	Comment
cin, cout	See section 1.7
Header File <ctype.h>	
Function	Comment
int tolower(int aCharacter)	Returns the lowercase of the letter
int toupper(int aCharacter)	Returns the uppercase of the letter
bool islower(int aCharacter)	Returns whether the letter is lower case
bool isalnum(int aCharacter)	Returns whether the character is alphanumeric
bool isalpha(int aCharacter)	Returns whether the character is alphabetic
bool isdigit(int aCharacter)	Returns whether the character is digit
bool isxdigit(int aCharacter)	Returns whether the character is hexadecimal digit
bool isspace(int aCharacter)	Returns whether the character is a white-space character
Header File <string.h>	
Function	Comment
void strcpy(String1, String2)	Copies <b>String2</b> to <b>String1</b>
void strncpy(String1, String2, n)	Copies <b>n</b> characters of <b>String2</b> to <b>Strin1</b>
void strcat(String1, String2)	Concatenates <b>String2</b> to <b>String1</b>
void strncat(String1, String2, n)	concatenates <b>n</b> characters of <b>String2</b> to <b>String1</b>
int strcmp(String1, String2)	returns negative integer if <b>String1</b> < <b>String2</b> , positive integer if <b>String1</b> > <b>String2</b> ; zero if <b>String1</b> == <b>String2</b>
int strncmp(String1, String2, n)	As for <b>strcmp</b> except that only <b>n</b> characters are considered.
char * strchr(String1, c)	Returns pointer to the first <b>c</b> in <b>String1</b> or NULL
char * strrchr(String1, c)	Returns pointer to the last <b>c</b> in <b>String1</b> or NULL
int strlen(aString)	Returns length of the string
strstr(String1, String2)	Returns pointer to the first <b>String2</b> in <b>String1</b> or NULL

Figure 4-2: Commonly Used Standard Functions (continued)

Header File <stdlib.h>	
Function	Comment
<code>float atof(aString)</code>	Converts a string to a double floating point number
<code>int atoi(aString)</code>	Converts a string to an integer
<code>long atol(aString)</code>	Converts a string to a long integer
<code>void itoa(Number, StringBuffer, Radix)</code>	Converts <b>Number</b> to string and stores result in <b>StringBuffer</b> , for the <b>Radix</b> specified.
<code>double rand( )</code>	Returns a random number between 0 and RAND_MAX, which is at least 32767
<code>void malloc(Size)</code>	Returns a pointer to a space for an object or NULL
<code>void free(Pointer)</code>	De-allocates the space pointed to by Pointer
Header File <conio.h>	
Function	Comment
<code>bool kbhit( )</code>	Returns <b>true</b> if a keyboard key is hit, <b>false</b> otherwise
Header File <time.h>	
The structure <i>tm</i> contains the following fields:	
Component	Comment
<code>tm_sec</code>	seconds after the minute
<code>tm_min</code>	minutes after the hour
<code>tm_hour</code>	hours since midnight
<code>tm_mday</code>	day of the month
<code>tm_mon</code>	months since January
<code>tm_year</code>	Years since 1900
<code>tm_wday</code>	days since Sunday
<code>tm_yday</code>	days since January 1
<code>tm_isdst</code>	Daylight Saving Flag; positive if DST; zero if not DST; negative if the information is not available.



## 4.4 Command Line Arguments to main ( )

Sometimes it is required to pass information to a program when it is called (from the operating system or another environment).

C++ has two built in (but optional) parameters to the **main ( )** function, namely **argc** and **argv**; they receive the command line arguments:

- The **argc** parameter is an integer that holds the number of arguments of the command line. Since the name of the program is also counted, it will always be at least 1.
- The **argv** parameter is a pointer to an array of character pointers (i.e. an array of null-terminated strings). Each item in the **argv** array points to a string containing the command line argument:  
argv[0] → the program name  
argv[1] → the program's first argument  
...  
argv[n] → the program's nth argument

The recommended declaration for a program using this command line is as follows:

```
void main (int argc, char *argv [ ])
{
    // ...
}
```

You can then refer to the arguments within your program by using the array subscripts.

**Example 4-6:** In the following example, the program name and first argument for a specific program are retrieved into variables **progName** and **studFile** respectively.

```
string progName, studFile; // C++ supports a string class; this will be covered later in the course
// ...
progName = argv[0];
studFile = argv[1];
```

In most (operating system) environments, argument separator is a space or tab. If an argument has spaces in it, the entire argument is enclosed in double quotes.

**Example 4-7:** The following sample operating system command line shows how you could call a program that expects arguments for those two parameters of the previous example. The program name is assumed to be **CP\_ProcessStud\_ECF**, and the file-name is assumed to be **StudentFile**.

```
> CP_ProcessStud_ECF "CP_ProcessStud_ECF" StudentFile
```

### 4.3 Command line Arguments to main ( ) (continued)

Numeric arguments must be converted from string to numeric form within the program.

**Example 4-8:** In the following code snippet, two integers are extracted from the program arguments.

```
int thisArg, otherArg;
// ...
thisArg = atoi (argv[1]);
otherArg = atoi (argv [2]);
```

### 4.5 Recursion

Recursion is the act of an algorithm (in the case of this course, a C++ function) invoking itself. Every recursion algorithm can be replaced by a non-recursive one; however, replacement with a non-recursive algorithm is not always trivial. C++ supports recursive functions.

#### 4.5.1 The Factorial Problem

As an example, consider the factorial of a number, defined as:

$$N! = N(N-1) (N-2) \dots (N-I +1) \dots (1)$$

Observe that for  $N > 1$ ,  $N! = N(N-1)!$  The recursive and non-recursive C++ functions are shown below.

**Example 4-9:** Below is a function to calculate  $N!$  via an iterative loop.

```
double getFactorial (int n)
{
    int x; double theResult;
    theResult =1;
    for (x =1; x<=n; x++) theResult = theResult * x;
    return theResult;
}
```

**Example 4-10:** Below is a function to calculate  $N!$  via a recursive call.

```
// Recursive version of N!
double getFactorial (int n)
{
    double theResult;
    if ((n==1) || (n == 0)) theResult =1;
    else theResult = n * getFactorial (n-1);
    return theResult;
}
```

### 4.5.2 The String Reversal Problem

String reversal is another classic case where recursion is applicable. Example 3-11 of the previous chapter provides you with two alternate iterative solutions to the string reversal problem. A third alternative is recursion. The pseudo-code is provided in figure 4-3; you are encouraged to write a C++ implementation of this algorithm on your own.

**Figure 4-3: Recursive String Reversal Algorithm**

**Algorithm: reverseS(inString) Returns a string**

```

Let thisString, revString be strings;
Let sLength be an integer;
/* Assume that there is a subroutine called Substring that returns the substring from a supplied string.
   For instance, Substring(thisString, start, length) returns a substring of length bytes from thisString,
   starting at start. Most programming languages have an implementation of this concept. */
START
  Determine sLength;
  If (sLength = 1)
    revString := thisString;
  Else
    revString := Substring(thisString, sLength, 1) + reverseS(Substring(thisString, 0, sLength - 1));
  End-If;
  Return revString;
STOP

```

### 4.5.3 Overarching Principle for Recursion

An overarching principle that governs all recursive problems is the mathematical principle of induction. To paraphrase, the principle asserts that if a phenomenon is governed by  $n$  consistent repetitions, that system will also conform to  $n + 1$  repetitions up to infinity, or  $n - 1$  repetitions all the way down to the simplest form.

When attempting to define a recursive solution to a problem, it is imperative that you identify the simplest form of the problem, and determine what action should take place. This is your *exit strategy*; without it, you'll wind up with an infinite loop. Next, you generalize from the exit strategy all the way up to  $n$  repetitions. On each repetition, determine the recursive action(s) to be taken; this is your *generalization strategy*.

Now revisit the factorial problem: Notice that your exit strategy is what happens when  $n$  is equal to zero or 1; at that point the factorial is 1. And your generalization strategy is what happens when  $n > 1$ . At that point, you simply multiply  $n$  by the factorial of  $n - 1$ .

Let's do a similar exercise with the string reversal problem: Your exit strategy is when the length of the string is 1; at that point, the reversed string is the string itself. Your generalization strategy is what happens when the length of the string is greater than 1. At that point, you simply peel off the last character and concatenate it to the reversal of the rest of the string.

## 4.6 Function Overloading

In C++, two or more functions can share the same name, as long as their parameters declarations and/or return types are different. The functions that share the same name are said to be *overloaded*; the process is referred to as *function overloading*.

Function overloading is one way C++ achieves polymorphism — a desirable feature of object oriented programming that will be discussed later in the course.

To overload a function, simply define different versions of it (by defining different parameter lists and/or return types). Depending on the argument supplied at function call, C++ will load the correct version of the function. The return type of each overload function may also be different, but that distinction alone is not sufficient for the compiler to distinguish the versions.

**Example 4-11:** Below are three overloaded functions.

```
/* Recall that the C header file <Math.h> has three distinct functions to return absolute
value: abs (...) for integer; labs (...) for long integer; and fabs (...) for double .
```

```
The reason for this is that C does not support function overloading. But since C++ does; we
could therefore have the following overloaded functions: */
```

```
int abs (int x)    // for integer
{
    if (x < 0)
        return -x;
    else return x;
}

long abs (long l)    // for long integer
{
    if (l < 0)
        return -l;
    else return l;
}

double abs (double d) // for double
{
    if (d < 0)
        return -d;
    else return d;
}
```

**Note:** The activity of overloaded functions need not relate to each other. However, it is strongly recommended that you write overloaded functions for closely related operations.

## 4.7 Default Function Arguments

You can specify default values for function parameter(s). When the function is called, if no argument is specified for a parameter with a default value, the default value is used as the argument.

Default parameter values are specified in a manner that is similar to variable initialization.

**Example4-12:** The following code illustrates the usefulness of default values on function parameters.

```
void displayMessage (char thisMessage [ ] = "Well done; take a bow")
{
    cout<< thisMessage;
}
// ...
// This function may be called in any of the following ways:
displayMessage( ); // The default message is displayed
displayMessage (myMessage); // The message in string myMessage is displayed
displayMessage ("Tough luck. Try again"); // The shown in double quotes is displayed
```

Following are some guidelines about using default parameters:

1. The default value for a parameter must be specified once — the first time the function is declared — either in the function prototype, or in the function definition. This means that if a prototype is used, the default values should be specified there; otherwise they must be specified at function definition.
2. You can specify different default values for different versions of an overloaded function. Bear in mind that each instance of the overloaded function has its own implementation at runtime.
3. All parameters that have default values must be declared after (i.e. to the right of) parameters that do not have default values.
4. Default values can act as shorthand forms of function overloading: If a function **F** requires **n** parameters in one form and down to **n-x** parameters in other scenarios (where  $x < n$ ), then rather than creating  $x$  versions of the overloaded function, we can assign default values for the **x** parameters that need them.

Whenever possible, this approach is preferred, since function overloading, coupled with automatic type conversion could lead to ambiguity.

## 4.8 Summary and Concluding Remarks

---

Here is a summary of what has been covered in this lecture:

- A subroutine (or subprogram) is a section of a computer program that is responsible for a specific task or set of related tasks. In C++, subprograms are implemented as functions.
- Many C++ environments require that a function prototype precede definition of the function. The function prototype is simply the heading of the function. It alerts the C++ compiler to expect the function.
- When a function is called, all supplied arguments are copied to its parameters in a positional manner.
- If the function returns a value, it must be called by including it in an expression. If it does not return a value, it can be called by simply stating its name, and any required argument(s) in a parenthesized list.
- C++ provides various special functions in their respective header files.
- C++ supports recursion — the ability of a function to call itself.
- Function overloading is the act of defining a function more than once, with each definition differing in parameters and/or return type, and possibly internal code.
- C++ also allows you to define functions that have default values assigned to their parameters. If a parameter has a default value, then specifying an argument for that parameter when the function is called is optional.

As we proceed in the course, you will more fully appreciate that writing functions is essential to programming in the C++ language; in fact, a C++ program is essentially a collection of one or more functions. The next lecture discusses *structures*, *pointers*, and *arrays*. Additionally, it will give you an opportunity to read and write more C++ functions.

## 4.9 Recommended Readings

---

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6<sup>th</sup> Edition. Boston: Addison-Wesley. See chapter 6.

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8<sup>th</sup> Edition. Boston: Pearson. See chapter 6.

[Kernighan & Richie 1988] Kernighan, Brian W. & Dennis M. Richie. 1988. *The C Programming Language*. Boston: Prentice Hall. See chapter 4.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5<sup>th</sup> Edition. Boston: Pearson. See chapters 3 & 4.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9<sup>th</sup> Edition. Boston: Pearson. See chapters 4 & 5.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapter 4.

---