

Lecture 9: Functions and Recursion – Part I

Class page: <https://github.com/tsung-wei-huang/cs1410-40>

Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT



Learning Objective

- To construct programs modularly from functions.
- To use common math library functions.
- The mechanisms for passing data to functions and returning results.
- The function call mechanism and activation records.
- To use random number generation to implement game-playing applications.
- How the visibility of identifiers is limited to specific regions of programs.
- To write recursive functions.

Introduction

- Construct programs from small, simple pieces, or components.
 - divide and conquer
- We emphasize **how to declare and use functions** to facilitate the design, implementation, operation and maintenance of large programs.
- We'll overview several C++ Standard Library math functions.
- You'll learn how to declare your own functions.
- We'll discuss function prototypes and how the compiler uses them to ensure that functions are called properly.
- We'll take a brief diversion into simulation techniques with random number generation and develop a version of the casino dice game called craps that uses most of the programming techniques you've learned.

Introduction

- Many of the applications you develop will have more than one function of the same name.
 - Function overloading
 - Used to implement functions that perform similar tasks for arguments of different types or possibly for different numbers of arguments.
- We consider function templates—a mechanism for defining a family of overloaded functions.
- The chapter concludes with a discussion of functions that call themselves, either directly, or indirectly through another function—a topic called recursion.

Introduction

- C++ programs are typically written by combining new functions and classes you write with “prepackaged” functions and classes available in the C++ Standard Library.
- The C++ Standard Library provides a rich collection of functions for
 - common mathematical calculations,
 - string manipulations,
 - character manipulations,
 - input/output,
 - error checking and
 - many other useful operations.

Introduction

- Functions you write are referred to as **user-defined functions** or **programmer-defined functions**.
- Motivations for “functionalizing” a program.
 - Divide-and-conquer makes program development more manageable.
 - Software reusability—using existing functions as building blocks to create new programs.
 - Programs can be created from standardized functions that accomplish specific tasks.
 - Avoid repeating code in a program.
 - Packaging code as a function allows the code to be executed from different locations in a program simply by calling the function.

Math Functions in <cmath>

Function	Description	Example
<code>ceil(x)</code>	rounds x to the smallest integer not less than x	<code>ceil(9.2)</code> is 10.0 <code>ceil(-9.8)</code> is -9.0
<code>cos(x)</code>	trigonometric cosine of x (x in radians)	<code>cos(0.0)</code> is 1.0
<code>exp(x)</code>	exponential function e^x	<code>exp(1.0)</code> is 2.718282 <code>exp(2.0)</code> is 7.389056
<code>fabs(x)</code>	absolute value of x	<code>fabs(5.1)</code> is 5.1 <code>fabs(0.0)</code> is 0.0 <code>fabs(-8.76)</code> is 8.76
<code>floor(x)</code>	rounds x to the largest integer not greater than x	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>fmod(x, y)</code>	remainder of x/y as a floating-point number	<code>fmod(2.6, 1.2)</code> is 0.2
<code>log(x)</code>	natural logarithm of x (base e)	<code>log(2.718282)</code> is 1.0 <code>log(7.389056)</code> is 2.0
<code>log10(x)</code>	logarithm of x (base 10)	<code>log10(10.0)</code> is 1.0 <code>log10(100.0)</code> is 2.0

Math Functions in <cmath>

Function	Description	Example
<code>pow(x, y)</code>	x raised to power y (x^y)	<code>pow(2, 7)</code> is 128 <code>pow(9, .5)</code> is 3
<code>sin(x)</code>	trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0
<code>sqrt(x)</code>	square root of x (where x is a nonnegative value)	<code>sqrt(9.0)</code> is 3.0
<code>tan(x)</code>	trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0

Function Definition

- The format of a function definition is as follows:

```
return-value-type function-name( parameter-list )  
{  
    declarations and statements  
}
```

- The *function-name* is any valid identifier.
- The *return-value-type* is the data type of the returned result to the caller.
 - The type `void` indicates that a function does not return a value.
- All variables defined in a function are **local variables**—they're known only in the function in which they're defined.
- Most functions have a list of **parameters** that provide the means for communicating information between functions.
 - A function's parameters are also local variables of that function.

Example

```
1 // Creating and using a programmer-defined function.
2 #include <iostream>
3 using namespace std;
4
5 int square( int ); // function prototype
6
7 int main()
8 {
9     // Loop 10 times and calculate and output the
10    // square of x each time
11    for ( int x = 1; x <= 10; x++ )
12        cout << square( x ) << " ";
13
14    cout << endl;
15 } // end main
16
17 // square function definition returns square of an integer
18 int square( int y ) // y is a copy of argument to function
19 {
20     return y * y;      // returns square of y as an int
21 } // end function square
```

Debrief

- Function **square** is **invoked** or **called** in **main** with the expression **square(x)** in line 13.
- The parentheses **()** in the function call are an operator in C++ that causes the function to be called.
- Function **square** (lines 19–22) receives a copy of the value of argument **x** from line 13 and stores it in the parameter **y**.
- Then **square** calculates **y * y** (line 21) and passes the result back to the point in **main** where **square** was invoked (line 13).
- The result is displayed.
- The function call does not change the value of **x**.
- The **for** repetition structure repeats this process for each of the values 1 through 10.

```
'  
8 int main()  
9 {  
10    // loop 10 times and calculate and output the  
11    // square of x each time  
12    for ( int x = 1; x <= 10; x++ )  
13        cout << square( x ) << " "; // function call  
14  
15    cout << endl;  
16 } // end main  
17  
18 // square function definition returns square of an integer  
19 int square( int y ) // y is a copy of argument to function  
20 {  
21    return y * y; // returns square of y as an int  
22 } // end function square
```

Debrief

- The definition of **square** (lines 19–22) shows that it uses integer parameter **y**.
- Keyword **int** preceding the function name indicates that **square** returns an integer result.
- The **return** statement in **square** (line 21) passes the result of the calculation back to the calling function.

```
8 int main()
9 {
10    // Loop 10 times and calculate and output the
11    // square of x each time
12    for ( int x = 1; x <= 10; x++ )
13        cout << square( x ) << " ";
14
15    cout << endl;
16 } // end main
17
18 // square function definition returns square of an integer
19 int square( int y ) // y is a copy of argument to function
20 {
21     return y * y;      // returns square of y as an int
22 } // end function square
```

Function With Multiple Arguments

```
1 // Finding the maximum of three floating-point numbers.
2 #include <iostream>
3 using namespace std;
4
5
6 double maximum( double, double, double ); // function prototype
7
8 int main()
9 {
10    double number1;
11    double number2;
12    double number3;
13
14    cout << "Enter three floating-point numbers: ";
15    cin >> number1 >> number2 >> number3;
16
17    // number1, number2 and number3 are arguments to
18    // the maximum function call
19    cout << "Maximum is: "
20    << maximum( number1, number2, number3 ) << endl;
21 } // end main
22
```

Function with Multiple Arguments

```
23 // function maximum definition;
24 // x, y and z are parameters
25 double maximum( double x, double y, double z )
26 {
27     double max = x; // assume x is largest
28
29     if ( y > max ) // if y is larger,
30         max = y; // assign y to max
31
32     if ( z > max ) // if z is larger,
33         max = z; // assign z to max
34
35     return max; // max is largest value
36 } // end function maximum
```

Function with Multiple Arguments

```
Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32
```

```
Enter three floating-point numbers: 1.1 3.333 2.22
Maximum is: 3.333
```

```
Enter three floating-point numbers: 27.9 14.31 88.99
Maximum is: 88.99
```

Case Study: Random Number Generator

- The element of chance can be introduced into computer applications by using the C++ Standard Library function **rand**.
 - The function prototype for the **rand** function is in **<cstdlib>**.
 - For example: **i = rand();**
- Function **rand** generates an unsigned integer between **0** and **RAND_MAX** (a constant defined in the **<cstdlib>** header file).
 - For GNU C++, the value of **RAND_MAX** is 2147483647; for Visual Studio, the value of **RAND_MAX** is 32767.
- To produce integers in the range **0** to **5**, we use the modulus operator (%) with **rand** as follows → **rand() % 6**
 - The number 6 is called the **scaling factor**.
- Shifting the range of numbers produces the integers from **1** to **6**.

1 + rand() % 6

Case Study: Random Number Generator

```
1 // Roll a six-sided die 6,000,000 times.
2 #include <iostream>
3 #include <iomanip>
4 #include <cstdlib> // contains function prototype for rand
5 using namespace std;
6
7 int main()
8 {
9     int frequency1 = 0; // count of 1s rolled
10    int frequency2 = 0; // count of 2s rolled
11    int frequency3 = 0; // count of 3s rolled
12    int frequency4 = 0; // count of 4s rolled
13    int frequency5 = 0; // count of 5s rolled
14    int frequency6 = 0; // count of 6s rolled
15
16    int face; // stores most recently rolled value
17
18    // summarize results of 6,000,000 rolls of a die
19    for ( int roll = 1; roll <= 6000000; roll++ )
20    {
21        face = 1 + rand() % 6; // random number from 1 to 6
22
23    }
```

Case Study: Random Number Generator

```
24 // determine roll value 1-6 and increment appropriate counter
25 switch ( face )
26 {
27     case 1:
28         ++frequency1; // increment the 1s counter
29         break;
30     case 2:
31         ++frequency2; // increment the 2s counter
32         break;
33     case 3:
34         ++frequency3; // increment the 3s counter
35         break;
36     case 4:
37         ++frequency4; // increment the 4s counter
38         break;
39     case 5:
40         ++frequency5; // increment the 5s counter
41         break;
42     case 6:
43         ++frequency6; // increment the 6s counter
44         break;
45     default: // invalid value
46         cout << "Program should never get here!";
47     } // end switch
48 } // end for
```

Case Study: Random Number Generator

```
49
50     cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
51     cout << "    1" << setw( 13 ) << frequency1
52         << "\n    2" << setw( 13 ) << frequency2
53         << "\n    3" << setw( 13 ) << frequency3
54         << "\n    4" << setw( 13 ) << frequency4
55         << "\n    5" << setw( 13 ) << frequency5
56         << "\n    6" << setw( 13 ) << frequency6 << endl;
57 } // end main
```

Face	Frequency
1	999702
2	1000823
3	999378
4	998898
5	1000777
6	1000422

True Randomness ...?

- ❑ Function `rand` actually generates **pseudorandom numbers**.
- ❑ The numbers in the sequence appear to be random, but the sequence repeats itself each time the program executes.
- ❑ It can be conditioned to produce a different sequence of random numbers for each execution.
- ❑ This is called **randomizing** and is accomplished with the C++ Standard Library function `srand`.
- ❑ Function `srand` takes an **unsigned integer argument** and **seeds** the `rand` function to produce a different sequence of random numbers for each execution.
 - ❑ The function prototype for `srand` is in header file `<cstdlib>`.

True Randomness ...?

```
1 // Randomizing die-rolling program.
2 #include <iostream>
3 #include <iomanip>
4 #include <cstdlib> // contains prototypes for functions srand and rand
5 using namespace std;
6
7 int main()
8 {
9     unsigned seed; // stores the seed entered by the user
10
11    cout << "Enter seed: ";
12    cin >> seed;
13    srand( seed ); // seed random number generator
14
15    // Loop 10 times
16    for ( int counter = 1; counter <= 10; counter++ )
17    {
18        // pick random number from 1 to 6 and output it
19        cout << setw( 10 ) << ( 1 + rand() % 6 );
20
21
```

True Randomness ...?

```
22      // if counter is divisible by 5, start a new line of output
23      if ( counter % 5 == 0 )
24          cout << endl;
25  } // end for
26 } // end main
```

Enter seed: 67

6	1	4	6	2
1	6	1	6	4

Enter seed: 432

4	6	3	1	6
3	1	5	4	2

Enter seed: 67

6	1	4	6	2
1	6	1	6	4

Practical Use of Seed

- To randomize without having to enter a seed each time, we may use a statement like

```
    srand( time( 0 ) );
```

- This causes the computer to read its clock to obtain the value for the seed.
- Function **time** (with the argument 0) typically returns the current time as **the number of seconds** since January 1, 1970, at midnight Greenwich Mean Time (GMT).
- This value is converted to an **unsigned integer** and used as the seed to the random number generator.
- The function prototype for **time** is in **<ctime>**.

Summary

- Function
- Function with Multiple Arguments
- Case Study: Random Number Generator