Lecture 12:
Functions

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 16th, 2010

6.6 C++ Standard Library Header Files

- The C++ Standard Library is divided into many portions, each with its own header file.
- The header files contain the function prototypes for the related functions that form each portion of the library.
- The header files also contain definitions of various class types and functions, as well as constants needed by those functions.
- A header file “instructs” the compiler on how to interface with library and user-written components.
- Figure 6.7 lists some common C++ Standard Library header files, most of which are discussed later in the book.

6.6 C++ Standard Library Header Files

Standard Library header file	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and standard output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output. This header file replaces header file `<iostream.h>`.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header file is first used in Section 4.9 and is discussed in more detail in Chapter 15, Stream Input/Output. This header file replaces header file `<iomanip.h>`.
`<cmath>`	Contains function prototypes for math library functions (discussed in Section 6.3). This header file replaces header file `<math.h>`.
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header file are covered in Section 6.7; Chapter 11, Operator Overloading; String and Array Objects; Chapter 16, Exception Handling; Chapter 21, Bits, Characters, C Strings and `structs`; and Appendix F, C Legacy Code Topics. This header file replaces header file `<stdlib.h>`.

Fig. 6.7 | C++ Standard Library header files. (Part 1 of 5.)

6.6 C++ Standard Library Header Files

Standard Library header file	Explanation
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header file replaces header file `<time.h>`. This header file is used in Section 6.7.
`<vector>`, `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these header files in Chapter 22, Standard Template Library (STL).
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file `<ctype.h>`. These topics are discussed in Chapter 21, Bits, Characters, C Strings and `structs`.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header file replaces header file `<string.h>`. This header file is used in Chapter 11, Operator Overloading; String and Array Objects.

Fig. 6.7 | C++ Standard Library header files. (Part 2 of 5.)

6.6 C++ Standard Library Header Files

Standard Library header file	Explanation
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8.
`<exception>`, `<stdexcept>`	These header files contain classes that are used for exception handling (discussed in Chapter 16, Exception Handling).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 16, Exception Handling.
`<fstream>`	Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 17, File Processing). This header file replaces header file `<fstream.h>`.
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 18, Class `string` and String Stream Processing).
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class `string` and String Stream Processing).
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header file is used in Chapter 22.

Fig. 6.7 | C++ Standard Library header files. (Part 3 of 5.)

6.6 C++ Standard Library Header Files

Standard Library header file	Explanation
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 22.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 22.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This replaces header file `<assert.h>` from pre-standard C++. This header file is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system. This header file replaces header file `<float.h>`.
`<climits>`	Contains the integral size limits of the system. This header file replaces header file `<limits.h>`.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions. This header file replaces header file `<stdio.h>`.
`<locale>`	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).

Fig. 6.7 | C++ Standard Library header files. (Part 4 of 5.)

6.6 C++ Standard Library Header Files

Standard Library header file	Explanation
`<limits>`	Contains classes for defining the numerical data type limits on each computer platform.
`<utility>`	Contains classes and functions that are used by many C++ Standard Library header files.

Fig. 6.7 | C++ Standard Library header files. (Part 5 of 5.)

6.7 Case Study: Random Number Generation

- The element of chance can be in-troduced into computer applications by using the C++ Standard Library function `rand`.
- The function `rand` generates an unsigned integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<cstdlib>` header file).
- The value of `RAND_MAX` must be at least 32767—the maximum positive value for a two-byte (16-bit) integer.
- For GNU C++, the value of `RAND_MAX` is 2147483647; for Visual Studio, the value of `RAND_MAX` is 32767.
- If `rand` truly produces integers at random, every number between 0 and `RAND_MAX` has an equal *chance (or probability) of being chosen each time* `rand` *is called.*

6.7 Case Study: Random Number Generation (cont.)

- The function prototype for the `rand` function is in `<cstdlib>`.
- To produce integers in the range 0 to 5, we use the modulus operator (%) with `rand`:
  - `rand() % 6`
  - This is called scaling.
  - The number 6 is called the scaling factor. Six values are produced.
- We can shift the range of numbers produced by adding a value.

6.7 Case Study: Random Number Generation (cont.)

```
1  // Fig. 6.8: fig06_08.cpp
2  // Shifted and scaled random integers.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains function prototype for rand
6  using namespace std;
7
8  int main()
9  {
10    // loop 20 times
11    for ( int counter = 1; counter <= 20; counter++ )
12    {
13       // pick random number from 1 to 6 and output it
14       cout << setw( 10 ) << ( 1 + rand() % 6 );
15
16       // if counter is divisible by 5, start a new line of output
17       if ( counter % 5 == 0 )
18          cout << endl;
19    } // end for
20 } // end main
```

Fig. 6.8 | Shifted, scaled integers produced by `1 + rand() % 6`. (Part 1 of 2.)

6.7 Case Study: Random Number Generation (cont.)

```
6  6  5  5  6
5  1  1  5  3
6  6  2  4  2
6  2  3  4  1
```

Fig. 6.8 | Shifted, scaled integers produced by `1 + rand() % 6`. (Part 2 of 2.)

6.7 Case Study: Random Number Generation (cont.)

- To show that the numbers produced by `rand` occur with approximately equal likelihood, Fig. 6.9 simu-lates 6,000,000 rolls of a die.
- Each integer in the range 1 to 6 should ap-pear approximately 1,000,000 times.

6.7 Case Study: Random Number Generation (cont.)

```
1  // Fig. 6.9: fig06_09.cpp
2  // Roll a six-sided die 6,000,000 times.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains function prototype for rand
6  using namespace std;
7
8  int main()
9  {
10    int frequency1 = 0; // count of 1s rolled
11    int frequency2 = 0; // count of 2s rolled
12    int frequency3 = 0; // count of 3s rolled
13    int frequency4 = 0; // count of 4s rolled
14    int frequency5 = 0; // count of 5s rolled
15    int frequency6 = 0; // count of 6s rolled
16
17    int face; // stores most recently rolled value
18
19    // summarize results of 6,000,000 rolls of a die
20    for ( int roll = 1; roll <= 6000000; roll++ )
21    {
22       face = 1 + rand() % 6; // random number from 1 to 6
23
```

Fig. 6.9 | Rolling a six-sided die 6,000,000 times. (Part 1 of 3.)

6.7 Case Study: Random Number Generation (cont.)

```
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
```

Fig. 6.9 | Rolling a six-sided die 6,000,000 times. (Part 2 of 3.)

6.7 Case Study: Random Number Generation (cont.)

```
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
```

Fig. 6.9 | Rolling a six-sided die 6,000,000 times. (Part 3 of 3.)

6.7 Case Study: Random Number Generation (cont.)

- Executing the program of Fig. 6.8 again produces exactly the same sequence of values.
  - This repeatability is an important characteristic of function `rand`.
  - Essential for proving that program works and for debugging.
- Function `rand` actually generates pseudorandom numbers.
  - Repeatedly calling `rand` produces a sequence of numbers that appears to be random.
  - The sequence repeats itself each time the program executes.

6.7 Case Study: Random Number Generation (cont.)

- Once a program has been debugged, 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.

6.7 Case Study: Random Number Generation (cont.)

```
1  // Fig. 6.10: fig06_10.cpp
2  // Randomizing die-rolling program.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contains prototypes for functions srand and rand
6  using namespace std;
7
8  int main()
9  {
10    unsigned seed; // stores the seed entered by the user
11
12    cout << "Enter seed: ";
13    cin >> seed;
14    srand( seed ); // seed random number generator
15
16    // loop 10 times
17    for ( int counter = 1; counter <= 10; counter++ )
18    {
19       // pick random number from 1 to 6 and output it
20       cout << setw( 10 ) << ( 1 + rand() % 6 );
21
```

Fig. 6.10 | Randomizing the die-rolling program. (Part 1 of 2.)

6.7 Case Study: Random Number Generation (cont.)

```
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
```

Fig. 6.10 | Randomizing the die-rolling program. (Part 2 of 2.)

6.7 Case Study: Random Number Generation (cont.)

- 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` as written in the preceding statement) typically re-turns the current time as the number of seconds since January 1, 1970, at midnight Greenwich Mean Time (GMT).
- The function prototype for `time` is in `<ctime>`.

6.7 Case Study: Random Number Generation (cont.)

- To produce random numbers in a specific range use:
  - `number = shiftingValue + rand() % scalingFactor;`
- where *shiftingValue is equal to the first number in the desired range of consecutive integers and scalingFactor is equal to the width of the de-sired range of consecutive integers.*

6.8 Case Study: Game of Chance; Introducing enum (cont.)

- An enumeration, introduced by the keyword `enum` and followed by a type name, is a set of integer constants represented by identifiers.
- The values of these enumeration constants start at `0`, unless specified otherwise, and increment by `1`.
- The identifiers in an `enum` must be unique, but separate enumeration constants can have the same integer value.
- Variables of an `enum` type can be assigned only one of the values declared in the enumeration.

6.8 Case Study: Game of Chance; Introducing enum (cont.)

- Another popular enumeration is
  - `enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };`
  - creates user-defined type `Months` with enumeration constants representing the months of the year.
  - The first value in the preceding enumeration is explicitly set to 1, so the remaining values increment from 1, resulting in the values 1 through 12.
- Any enumeration constant can be assigned an integer value in the enumeration definition.

6.8 Case Study: Game of Chance; Introducing enum (cont.)

Good Programming Practice 6.3

Using enumerations rather than integer constants can make programs clearer. You can set the value of an enumeration constant once in the enumeration declaration.

6.9 Storage Classes (cont.)

- Local variables declared `static` are still known only in the function in which they're declared, but, unlike automatic variables, `static` local variables retain their values when the function returns to its caller.
- The next time the function is called, the `static` local variables contain the values they had when the function last completed execution.

6.10 Scope Rules

- The portion of the program where an identifier can be used is known as its scope.
- Four scopes for an identifier
  - function scope,
  - global namespace scope,
  - local scope and
  - function-prototype scope.

6.10 Scope Rules (cont.)

- An identifier declared outside any function or class has global namespace scope.
  - Global variables, function definitions and function prototypes placed outside a function
- Identifiers declared inside a block have local scope.
  - Local scope begins at the identifier's declaration and ends at the terminating right brace (}) of the block in which the identifier is declared.
  - Local vari-ables and function parameters.

6.10 Scope Rules (cont.)

- Any block can contain variable declara-tions.
- When blocks are nested and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is “hidden” until the inner block terminates.
- Local variables declared `static` still have local scope, even though they exist from the time the program begins execution.
- The only identifiers with function prototype scope are those used in the parameter list of a function prototype.

6.10 Scope Rules (cont.)

Common Programming Error 6.9

Accidentally using the same name for an identifier in an inner block that is used for an identifier in an outer block, when in fact you want the identifier in the outer block to be active for the duration of the inner block, is typically a logic error.

6.10 Scope Rules (cont.)

```
1  // Fig. 6.12: fig06_12.cpp
2  // A scoping example.
3  #include <iostream>
4  using namespace std;
5
6  void useLocal(); // function prototype
7  void useStaticLocal(); // function prototype
8  void useGlobal(); // function prototype
9
10 int x = 1; // global variable
11
12 int main()
13 {
14    cout << "global x in main is " << x << endl;
15
16    int x = 5; // local variable to main
17
18    cout << "local x in main's outer scope is " << x << endl;
19
20    { // start new scope
21       int x = 7; // hides both x in outer scope and global x
22
23       cout << "local x in main's inner scope is " << x << endl;
24    } // end new scope
```

Fig. 6.12 | Scoping example. (Part 1 of 4.)

6.11 Function Call Stack and Activation Records

- To understand how C++ performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a stack.
- Analogous to a pile of dishes.
  - When a dish is placed on the pile, it's normally placed at the top (referred to as pushing).
  - Similarly, when a dish is removed from the pile, it's normally removed from the top (referred to as popping).
- Last-in, first-out (LIFO) data structures—the last item pushed (inserted) is the first item popped (removed).

6.11 Function Call Stack and Activation Records (cont.)

- One of the most important mechanisms for computer science students to understand is the function call stack (or program execution stack).
  - supports the function call/return mechanism.
  - Also supports the creation, maintenance and destruction of each called function's automatic variables.
- Each function eventually must return control to the function that called it.
- Each time a function calls another function, an entry is pushed onto the function call stack.
  - This entry, called a stack frame or an activation record, contains the return address that the called function needs in order to return to the calling function.

6.11 Function Call Stack and Activation Records (cont.)

- When a function call returns, the stack frame for the function call is popped, and control transfers to the return address in the popped stack frame.
- Stack frames also maintain the memory for local automatica variables.

6.11 Function Call Stack and Activation Records (cont.)

```
1  // Fig. 6.13: fig06_13.cpp
2  // square function used to demonstrate the function
3  // call stack and activation records.
4  #include <iostream>
5  using namespace std;
6
7  int square( int ); // prototype for function square
8
9  int main()
10 {
11    int a = 10; // value to square (local automatic variable in main)
12
13    cout << a << " squared: " << square( a ) << endl; // display a squared
14 } // end main
15
16 // returns the square of an integer
17 int square( int x ) // x is a local variable
18 {
19    return x * x; // calculate square and return result
20 } // end function square
```

Fig. 6.13 | `square` function used to demonstrate the function call stack and activation records.

6.11 Function Call Stack and Activation Records (cont.)

```
10 squared: 100
```

Fig. 6.13 | `square` function used to demonstrate the function call stack and activation records.

6.12 Functions with Empty Parameter Lists

- In C++, an empty parameter list is specified by writing either `void` or nothing at all in parentheses.

6.12 Functions with Empty Parameter Lists

```
1  // Fig. 6.17: fig06_17.cpp
2  // Functions that take no arguments.
3  #include <iostream>
4  using namespace std;
5
6  void function1(); // function that takes no arguments
7  void function2( void ); // function that takes no arguments
8
9  int main()
10 {
11    function1(); // call function1 with no arguments
12    function2(); // call function2 with no arguments
13 } // end main
14
15 // function1 uses an empty parameter list to specify that
16 // the function receives no arguments
17 void function1()
18 {
19    cout << "function1 takes no arguments" << endl;
20 } // end function1
21
```

Fig. 6.17 | Functions that take no arguments. (Part 1 of 2.)

6.12 Functions with Empty Parameter Lists

```
22  // function2 uses a void parameter list to specify that
23  // the function receives no arguments
24  void function2( void )
25  {
26     cout << "function2 also takes no arguments" << endl;
27  } // end function2
```

```
function1 takes no arguments
function2 also takes no arguments
```

Fig. 6.17 | Functions that take no arguments. (Part 2 of 2.)

6.13 Inline Functions

- C++ provides inline functions to help reduce function call overhead—especially for small functions.
- Placing the qualifier `in-line` before a function's return type in the function definition "advises" the compiler to generate a copy of the function's code in place (when appropriate) to avoid a function call.
- Trade-off
  - Multiple copies of the function code are inserted in the program (often making the program larger) rather than there being a single copy of the function to which control is passed each time the function is called.
- The complete function definition must appear before the code is inlined so that the compiler knows how to expand a function call into its inlined code.

6.13 Inline Functions

Software Engineering Observation 6.10

Any change to an `inline` *function requires all clients of the function to be recompiled. This can be significant in some program development and maintenance situations.*

6.13 Inline Functions

Good Programming Practice 6.5

The `inline` qualifier should be used only with small, frequently used functions.

6.13 Inline Functions

Performance Tip 6.4

Using `inline` *functions can reduce execution time but may increase program size.*

6.13 Inline Functions

```
1  // Fig. 6.18: fig06_18.cpp
2  // Using an inline function to calculate the volume of a cube.
3  #include <iostream>
4  using namespace std;
5
6  // Definition of inline function cube. Definition of function appears
7  // before function is called, so a function prototype is not required.
8  // First line of function definition acts as the prototype.
9  inline double cube( const double side )
10 {
11    return side * side * side; // calculate cube
12 } // end function cube
13
14 int main()
15 {
16    double sideValue; // stores value entered by user
17    cout << "Enter the side length of your cube: ";
18    cin >> sideValue; // read value from user
19
20    // calculate cube of sideValue and display result
21    cout << "Volume of cube with side "
22       << sideValue << " is " << cube( sideValue ) << endl;
23 } // end main
```

Fig. 6.18 | `inline` function that calculates the volume of a cube. (Part 1 of 2.)

6.13 Inline Functions

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Fig. 6.18 | `inline` function that calculates the volume of a cube. (Part 2 of 2.)

Questions