Chapter 5 Functions and an Introduction to Recursion

Yu-Shuen Wang, CS, NCTU

5.7 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.
  - 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.

Standard Library header file	Explanation
`<iostream>`	Contains function prototypes for the standard input and standard output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header file is first used in Section 3.9 and is discussed in more detail in Chapter 15, Stream Input/Output.
`<cmath>`	Contains function prototypes for math library functions (discussed in Section 5.3).
`<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 5.8; Chapter 11, Operator Overloading; Chapter 16, Exception Handling; Chapter 22, Bits, Characters, C Strings and `structs`; and Appendix F, C Legacy Code Topics.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header file is used in Section 5.9.

Fig. 5.6 | C++ Standard Library header files. (Part 1 of 4.)

Standard Library header file	Explanation
`<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 6, Arrays and Vectors. We discuss all these header files in Chapter 21, 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. These topics are discussed in Chapter 22, Bits, Characters, C Strings and `structs`.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header file is used in Chapter 11, Operator Overloading.
`<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).

Fig. 5.6 | C++ Standard Library header files. (Part 2 of 4.)

Standard Library header file	Explanation
`<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, Random-Access Files).
`<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 21.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 21.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 21.

Fig. 5.6 | C++ Standard Library header files. (Part 3 of 4.)

Standard Library header file	Explanation
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This header file is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions.
`<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.).
`<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. 5.6 | C++ Standard Library header files. (Part 4 of 4.)

5.15 References and Reference Parameters

- Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference.
- When an argument is passed by value, a *copy* of the argument's value is made and passed to the called function.
- Changes to the copy do not affect the original variable's value in the caller.
- This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems.

Performance Tip 5.5

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

5.15 References and Reference Parameters (cont.)

- reference parameters—the first of the two means C++ provides for performing pass-by-reference.
- With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data.
- A reference parameter is an alias for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (&).

Performance Tip 5.6

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

Software Engineering Observation 5.12

Pass-by-reference can weaken security; the called function can corrupt the caller's data.

5.15 References and Reference Parameters (cont.)

- Without checking the function prototypes or function definitions, it isn't possible to tell from the calls alone whether either function can modify its arguments.

Common Programming Error 5.16

Because reference parameters are mentioned only by name in the body of the called function, you might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original variables are changed by the function.

Performance Tip 5.7

For passing large objects, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.

```
1  // Fig. 5.18: fig05_18.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using namespace std;
5
6  int squareByValue( int ); // function prototype (value pass)
7  void squareByReference( int & ); // function prototype (reference pass)
8
9  int main()
10 {
11    int x = 2; // value to square using squareByValue
12    int z = 4; // value to square using squareByReference
13
14    // demonstrate squareByValue
15    cout << "x = " << x << " before squareByValue\n";
16    cout << "Value returned by squareByValue: "
17       << squareByValue( x ) << endl;
18    cout << "x = " << x << " after squareByValue\n" << endl;
19
20    // demonstrate squareByReference
21    cout << "z = " << z << " before squareByReference" << endl;
22    squareByReference( z );
23    cout << "z = " << z << " after squareByReference" << endl;
24 } // end main
```

Fig. 5.18 | Passing arguments by value and by reference. (Part 1 of 2.)

```
25
26 // squareByValue multiplies number by itself, stores the
27 // result in number and returns the new value of number
28 int squareByValue( int number )
29 {
30    return number *= number; // caller's argument not modified
31 } // end function squareByValue
32
33 // squareByReference multiplies numberRef by itself and stores the result
34 // in the variable to which numberRef refers in function main
35 void squareByReference( int &numberRef )
36 {
37    numberRef *= numberRef; // caller's argument modified
38 } // end function squareByReference
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 5.18 | Passing arguments by value and by reference. (Part 2 of 2.)

5.15 References and Reference Parameters (cont.)

- To specify a reference to a constant, place the `const` qualifier before the type specifier in the parameter declaration.

- References can also be used as aliases for other variables within a function.
  - For example, the code

```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for
count
cRef++; // increment count (using its alias cRef)
```

  increments variable `count` by using its alias `cRef`.

5.15 References and Reference Parameters (cont.)

- Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables.

- Once a reference is declared as an alias for another variable, all operations performed on the alias are actually performed on the original variable.

```
1  // Fig. 5.19: fig05_19.cpp
2  // Initializing and using a reference.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 3;
9     int &y = x; // y refers to (is an alias for) x
10
11    cout << "x = " << x << endl << "y = " << y << endl;
12    y = 7; // actually modifies x
13    cout << "x = " << x << endl << "y = " << y << endl;
14 } // end main
```

```
x = 3
y = 3
x = 7
y = 7
```

Fig. 5.19 | Initializing and using a reference.

```
1  // Fig. 5.20: fig05_20.cpp
2  // References must be initialized.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 3;
9     int &y; // Error: y must be initialized
10
11    cout << "x = " << x << endl << "y = " << y << endl;
12    y = 7;
13    cout << "x = " << x << endl << "y = " << y << endl;
14 } // end main
```

Fig. 5.20 | Uninitialized reference causes a compilation error. (Part 1 of 2.)

5.15 References and Reference Parameters (cont.)

- Functions can return references, but this can be dangerous.
- When returning a reference to a variable declared in the called function, the variable should be declared `static` in that function.
- Otherwise, the reference refers to an automatic variable that's discarded when the function terminates.

Common Programming Error 5.17

Returning a reference to an automatic variable in a called function is a logic error. Some compilers issue a warning when this occurs.

5.16 Default Arguments

- You can specify that such a parameter has a default argument, i.e., a default value to be passed to that parameter.
- When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument.

5.16 Default Arguments (cont.)

- Default arguments must be specified with the first occurrence of the function name—typically, in the function prototype.
- Default values can be any expression, including constants, global variables or function calls.

```
1  // Fig. 5.21: fig05_21.cpp
2  // Using default arguments.
3  #include <iostream>
4  using namespace std;
5
6  // function prototype that specifies default arguments
7  int boxVolume( int length = 1, int width = 1, int height = 1 );
8
9  int main()
10 {
11    // no arguments--use default values for all dimensions
12    cout << "The default box volume is: " << boxVolume();
13
14    // specify length; default width and height
15    cout << "\n\nThe volume of a box with length 10,\n"
16       << "width 1 and height 1 is: " << boxVolume( 10 );
17
18    // specify length and width; default height
19    cout << "\n\nThe volume of a box with length 10,\n"
20       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
21
```

Fig. 5.21 | Default arguments to a function. (Part 1 of 2.)

```
22       // specify all arguments
23       cout << "\n\nThe volume of a box with length 10,\n"
24          << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
25          << endl;
26    } // end main
27
28    // function boxVolume calculates the volume of a box
29    int boxVolume( int length, int width, int height )
30    {
31       return length * width * height;
32    } // end function boxVolume
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 5.21 | Default arguments to a function. (Part 2 of 2.)

Good Programming Practice 5.8

Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.

5.18 Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different signatures.
- This is called function overloading.
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.
- Function overloading is used to create several functions of the same name that perform similar tasks, but on different data types.
- Function `main` cannot be overloaded.

Good Programming Practice 5.10

Overloading functions that perform closely related tasks can make programs more readable and understandable.

```
1  // Fig. 5.23: fig05_23.cpp
2  // Overloaded functions.
3  #include <iostream>
4  using namespace std;
5
6  // function square for int values
7  int square( int x )
8  {
9     cout << "square of integer " << x << " is ";
10    return x * x;
11 } // end function square with int argument
12
13 // function square for double values
14 double square( double y )
15 {
16    cout << "square of double " << y << " is ";
17    return y * y;
18 } // end function square with double argument
19
```

Fig. 5.23 | Overloaded `square` functions. (Part 1 of 2.)

```
20  int main()
21  {
22     cout << square( 7 ); // calls int version
23     cout << endl;
24     cout << square( 7.5 ); // calls double version
25     cout << endl;
26  } // end main
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Fig. 5.23 | Overloaded `square` functions. (Part 2 of 2.)

Common Programming Error 5.19

Creating overloaded functions with identical parameter lists and different return types is a compilation error.

```
1  // Fig. 5.24: fig05_24.cpp
2  // Name mangling.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 } // end function square
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20    // empty function body
21 } // end function nothing1
22
```

Fig. 5.24 | Name mangling to enable type-safe linkage. (Part 1 of 2.)

```
23  // function that receives arguments of types
24  // char, int, float & and double &
25  int nothing2( char a, int b, float &c, double &d )
26  {
27     return 0;
28  } // end function nothing2
29
30  int main()
31  {
32  } // end main
```

```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
_main
```

Fig. 5.24 | Name mangling to enable type-safe linkage. (Part 2 of 2.)

i: int, c: char, f: float, Rf: float &, Rd: double &

5.18 Function Overloading (cont.)

- The compiler uses only the parameter lists to distinguish between overloaded functions.
- Such functions need not have the same number of parameters.
- Use caution when overloading functions with default parameters, because this may cause ambiguity.

Common Programming Error 5.20

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having a program that contains both a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to use that function name in a call passing no arguments. The compiler does not know which version of the function to choose.

```
int boxvolume(int width=1, int height=1, int depth=1);

int boxvolume();
```

5.19 Function Templates

- Overloaded functions are normally used to perform similar operations that involve different program logic on different data types.
- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.

```
1  // Fig. 5.25: maximum.h
2  // Definition of function template maximum.
3  template < class T >  // or template< typename T >
4  T maximum( T value1, T value2, T value3 )
5  {
6     T maximumValue = value1; // assume value1 is maximum
7
8     // determine whether value2 is greater than maximumValue
9     if ( value2 > maximumValue )
10       maximumValue = value2;
11
12    // determine whether value3 is greater than maximumValue
13    if ( value3 > maximumValue )
14       maximumValue = value3;
15
16    return maximumValue;
17 } // end function template maximum
```

Fig. 5.25 | Function template `maximum` header file.

5.19 Function Templates (cont.)

- The formal type parameters are placeholders for fundamental types or user-defined types.
- These placeholders are used to specify the types of the function's parameters (line 4), to specify the function's return type (line 4) and to declare variables within the body of the function definition (line 6).
- A function template is defined like any other function, but uses the formal type parameters as placeholders for actual data types.

5.19 Function Templates (cont.)

- The name of a type parameter must be unique in the template parameter list for a particular template definition.
- When the compiler detects a `maximum` invocation in the program source code, the type of the data passed to `maximum` is substituted for `T` throughout the template definition, and C++ creates a complete function for determining the maximum of three values of the specified data type—all three must have the same type.
- Templates are a means of code generation.

```
1  // Fig. 5.26: fig05_26.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  #include "maximum.h" // include definition of function template maximum
5  using namespace std;
6
7  int main()
8  {
9     // demonstrate maximum with int values
10    int int1, int2, int3;
11
12    cout << "Input three integer values: ";
13    cin >> int1 >> int2 >> int3;
14
15    // invoke int version of maximum
16    cout << "The maximum integer value is: "
17       << maximum( int1, int2, int3 );
18
19    // demonstrate maximum with double values
20    double double1, double2, double3;
21
22    cout << "\n\nInput three double values: ";
23    cin >> double1 >> double2 >> double3;
24
```

Fig. 5.26 | Demonstrating function template `maximum`. (Part 1 of 2.)

```
25      // invoke double version of maximum
26      cout << "The maximum double value is: "
27         << maximum( double1, double2, double3 );
28
29      // demonstrate maximum with char values
30      char char1, char2, char3;
31
32      cout << "\n\nInput three characters: ";
33      cin >> char1 >> char2 >> char3;
34
35      // invoke char version of maximum
36      cout << "The maximum character value is: "
37         << maximum( char1, char2, char3 ) << endl;
38   } // end main
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 5.26 | Demonstrating function template `maximum`. (Part 2 of 2.)