Chapter 6

Functions

6.1 User Defined Functions

In order to make your programming task more structured and to promote code reuse, it is common to write your own subroutines, called functions. It is common to include these functions in the same source file as the `main()` function. Let's start with a function that returns the larger of two integers.

```
int larger(int i, int j){
    return i > j ? i : j;
}
```

We would want to use this function in a program like this:

```
cout << "The larger of " << i << " and " << j << " is: "
     << larger(i, j);
```

When the compiler reaches this call to the `larger` function, it must verify that there does indeed exist a function named `larger` that takes two integer parameters. For this reason, we must include a *function prototype* in or source file at a point before the function call appears. The function prototype consists of the function signature only:

```
int larger(int, int);
```

Although not required, it is common to include names for the individual parameters, like this:

```
int larger(int i, int j);
```

As an example of code reuse, we can use our `larger` function to construct a new function that returns the larger of three integers. This is also an example of function overloading, in which two different functions have the same name, but differ in the number or type of parameters. This new `larger` function looks like this:

```
int larger(int i, int j, int k){
    return larger(i, larger(j, k));
}
```

A complete program using these function is shown in Listing 6.1.

```
// File: Larger.cpp

#include <iostream>
using namespace std;

// Function prototypes, overloaded functions
int larger(int i, int j);
int larger(int i, int j, int k);

int main(){
   int i, j, k;
   cout << "Enter two integers: ";
   cin >> i >> j;

   cout << "The larger of " << i
      << " and " << j << " is: "
      << larger(i, j) << endl << endl;

   cout << "Now enter one more integer: ";
   cin >> k;

   cout << "The larger of " << i
      << ", " << j << " and " << k
      << " is: " << larger(i, j, k)
      << endl << endl;

   return EXIT_SUCCESS;
}

// Function implementations
int larger(int i, int j){
    return i > j ? i : j;
}

int larger(int i, int j, int k){
    return larger(i, larger(j, k));
}
```

Listing 6.1: Larger.cpp

6.2 Scope Rules

There are two type of identifiers in C++: local identifiers and global identifiers.

1. Local Identifiers - Declared inside a block `{}` or a function. These identifiers are not accessible outside of that block or function in which they are declared.

2. Global Identifiers - Declared outside all blocks and functions. These identifiers are accessible from anywhere in the program. It is a good idea to avoid using global identifiers since they lead to a high degree of coupling. Listing 6.2 and Figure 6.1 show an example of how global variables behave.

```cpp
// File: global1.cpp

#include <iostream>
using namespace std;

int g; // global variable

void func();

int main() {
      g = 10;
      cout << "in main global is " << g << endl;
      func();
      cout << "back in main global is " << g << endl;

      return EXIT_SUCCESS;
}

void func() {
      // Global accessable from with in function
      cout << "inside function global is " << g << endl;
      g++;
      cout << "new value of global is " << g << endl;
      return;
}
```

Listing 6.2: global1.cpp

```
in main global is 10
inside function global is 10
new value of global is 11
back in main global is 11
```

Figure 6.1: Output of global1.cpp

Whenever a variable is created, it is placed in a particular *scope*. It is illegal to have two variables with the same name in the same scope. However, two variables that are in different scopes can have the same name. When you refer to a variable name in your program, the following rules are used to decide which variable should be accessed.

1. Global variables can be accessed from inside a function provided both of the following are true:
   (a) The global identifier is declared *before* the function is implemented. The variable declaration can come before or after the function prototype, but it must come before the function implementation.
   (b) The global variable is not *hidden*. The following things can hide a global identifier
   - a function with the same name as the global identifier
   - a parameter with the same name as the global identifier (see Listing 6.3 and Figure 6.2)

```
// File global2.cpp

#include <iostream>
using namespace std;

int g = 19;
void increment(int g);

int main() {

      cout << "in main, g is " << g << endl;
      increment(g);
      cout << "back in main, g is still " << g << endl;

      return EXIT_SUCCESS;
}

void increment(int g) {
      g++; // Local copy, global copy is hidden by parameter
      cout << "in function, g is " << g << endl;
      return;
}
```

Listing 6.3: global2.cpp

```
in main, g is 19
in function, g is 20
back in main, g is still 19
```

Figure 6.2: Output of global2.cpp

- a local identifier with the same name as the global identifier (see Listing 6.4 and Figure 6.3)

```cpp
// File: global3.cpp

#include <iostream>
using namespace std;

int g = 19;
void increment(int val);

int main() {
      int x = 39, y = 92;

      cout << "in main, g is " << g << endl;
      increment(g);
      cout << "back in main, g is still " << g << endl;

      return EXIT_SUCCESS;
}

void increment(int val) {
      int g = val; // Local variable, hides global variable
      g++;
      cout << "in function, g is " << g << endl;
      return;
}
```

Listing 6.4: global3.cpp

```
in main, g is 19
in function, g is 20
back in main, g is still 19
```

Figure 6.3: Output of global3.cpp

2. Inside a nested block, an identifier can be accessed
   - Only from within that block, from the point of declaration forward
   - From within more deeply nested blocks, provided it is not hidden inside those blocks

Listing 6.5 and Figure 6.4 show an example of block scope. Pay particular attention to the unary *scope resolution operator* ::.

```
// File: scope.cpp

#include <iostream>
using namespace std;

void func();

int x = 7; // global scope
int main() {
   cout << "global, x = " << x << endl;

   int x = 70; // local scope in main
   cout << "local in main, x = " << x << endl;
   {
      int x = 700; // block scope
      cout << "block, x = " << x << endl;
   }
   cout << "local in main, x = " << x << endl;
    cout << "global, ::x = " << ::x << endl;

    func();

   return EXIT_SUCCESS;
}

void func(){
    int x = 7000;
    cout << "local in function, x = " << x << endl;
    cout << "use :: to access global variable, ::x = " << ::x << endl;
}
```

Listing 6.5: scope.cpp

```
global, x = 7
local in main, x = 70
block, x = 700
local in main, x = 70
global, ::x = 7
local in function, x = 7000
use :: to access global variable, ::x = 7
```

Figure 6.4: Output of scope.cpp

6.3 Default Arguments

You can specify default values for the argument of your functions. If a function call is made with some of the arguments omitted, then the default values are used. The arguments with default values must be the right most arguments in the function signature. Listing 6.6 and Figure 6.5 show an example of using default function argument.

```
// File: defaultArgs.cpp

#include <iostream>
using namespace std;

double cubeVolume(double length = 1.0,
                  double width = 2.0,
                  double height = 3.0);

int main(){
    cout << "Specify L=10, W=20, H=30. Volume = "
        << cubeVolume(10, 20, 30) << endl;

    cout << "Specify L=10, W=20. Volume = "
        << cubeVolume(10,20) << endl;

    cout << "Specify L=10. Volume =  "
        << cubeVolume(10) << endl;

    cout << "Use all defaults. Volume = "
        << cubeVolume() << endl;

    return EXIT_SUCCESS;
}

double cubeVolume(double length, double width, double height) {
    return length * width * height;
}
```

Listing 6.6: defaultArgs.cpp

```
Specify L=10, W=20, H=30. Volume = 6000
Specify L=10, W=20. Volume = 600
Specify L=10. Volume =  60
Use all defaults. Volume = 6
```

Figure 6.5: Output of defaultArgs.cpp

6.4 Function Overloading

C++ allows you two write multiple functions with the same name. You just need to make sure that either the *number* of parameters is different in each function, or that the *type* of the parameters is different in each function. Suppose your program is using `int`, `long`, `float` and `double` variables. And suppose you want a function to compute the square of each of theses variables. Without function overloading you would need to have the following functions:

```
int squareInt(int x);
long squareLong(long x);
float squareFloat(float x);
double squareDouble(double x);
```

Function overloading allows you to use the same name for each of these four functions (since the type of the parameters is different). You can now have these four functions:

```
int square(int x);
long square(long x);
float square(float x);
double square(double x);
```

Listing 6.7 and Figure 6.6 show this example in full.

```
// File: overloading.cpp

#include <iostream>
using namespace std;

int square(int x);
long square(long x);
float square(float x);
double square(double x);

int main(){
    int iValue = 2;
    long lValue = 4;
    float fValue = 6;
    double dValue = 8;

    cout << "square(iValue) = " << square(iValue) << endl
         << "square(lValue) = " << square(lValue) << endl
         << "square(fValue) = " << square(fValue) << endl
         << "square(dValue) = " << square(dValue) << endl;

    return EXIT_SUCCESS;
}

int square(int x){ return x * x; }
long square(long x){ return x * x; }
```

```
float square(float x){ return x * x; }
double square(double x){ return x * x; }
```

Listing 6.7: overloading.cpp

```
square(iValue) = 4
square(lValue) = 16
square(fValue) = 36
square(dValue) = 64
```

Figure 6.6: Output of overloading.cpp

6.5 Reference Parameters

When a parameters passed to a function, either the *value* of the parameter can be passed, or the *memory address* of the parameter can be passed. When the value is used, we say the parameter is *pass by value*, when the memory address is used, we say the parameter is *pass by reference*. To find out what memory address in which a variable is stores, you use the & operator, as shown in Listing 6.8 and Figure 6.7.

```
// File: references.cpp

# include <iostream>
using namespace std;

int main() {
    int x = 17;
    cout << "The value of x is " << x << endl;
    cout << "The memory address of x is " << &x << endl;

    return EXIT_SUCCESS;
}
```

Listing 6.8: references1.cpp

```
The value of x is 17
The memory address of x is 003AF934
```

Figure 6.7: Output of references1.cpp

When value parameters are used, the actual parameter cannot be changed by the function. However, when reference parameters are used, the value of the actual parameter can be changed. This is shown in Listing 6.9 and Figure 6.8. There are several things to note about this program. In order to tell a function to use pass by reference, you place the `&` symbol into the function signature, as is done on lines 7 and 34. However, the function calls for pass by value and for pass by reference are identical (see lines 15 and 21). Also note that the bodies of the two functions are identical.

```
 1  // File: references2.cpp
 2
 3  #include <iostream>
 4  using namespace std;
 5
 6  int squareByValue(int x);
 7  int squareByReference(int &x);
 8
 9  int main(){
10      int number = 8;
11      cout << "number started off equal to "
12           << number << endl;
13
14      cout << "number squared by value = "
15           << squareByValue(number) << endl;
16
17      cout << "number is still equal to "
18           << number << endl;
19
20      cout << "number squared by reference = "
21           << squareByReference(number) << endl;
22
23      cout << "now number has been changed to "
24           << number << endl;
25
26      return EXIT_SUCCESS;
27  }
28
29  int squareByValue(int x){
30      x = x * x;
31      return x;
32  }
33
34  int squareByReference(int &x){
35      x = x * x;
36      return x;
37  }
```

Listing 6.9: references2.cpp

```
number started off equal to 8
number squared by value = 64
number is still equal to 8
number squared by reference = 64
now number has been changed to 64
```

Figure 6.8: Output of references2.cpp

Reference parameters are often used when you wish to have a function return more than one value. Listing 6.10 and Figure 6.9 show a function that calculates both the square of a number a the double of a number.

```cpp
// File: references3.cpp

#include <iostream>
using namespace std;

void squareAndDouble(int x, int & xSquared, int & xDoubled);

int main(){
    int number = 8, squared, doubled;

    squareAndDouble(number, squared, doubled);

    cout << "number = " << number << endl
         << "squared = " << squared << endl
         << "doubled = " << doubled << endl;

    return EXIT_SUCCESS;
}

void squareAndDouble(int x, int & xSquared, int & xDoubled){
    xSquared = x * x;
    xDoubled = 2 * x;
}
```

Listing 6.10: references3.cpp

```
number = 8
squared = 64
doubled = 16
```

Figure 6.9: Output of references3.cpp

If you have a large object that you want to pass to a function, using pass by reference can really improve the performance of your program. However, you may not want to allow the function to modify the object. This can be done by using the `const` keyword as shown in Listing 6.11.

```
// File: references4.cpp

#include <iostream>
using namespace std;

int squareByRef(const int &x);

int main(){
     int number = 9;
     cout << "number squared = "
          << squareByRef(number) << endl;
     return EXIT_SUCCESS;
}

int squareByRef(const int &x){
     // Any attempt to change the value of x
     // results in a compilation error.
     int newValue = x * x;
     return newValue;
}
```

Listing 6.11: references4.cpp

References can also be used to create an alias for a variable. For example, in Listing 6.12 and Figure 6.10 variable `y` is created as an alias for variable `x`. Both of these variables refer to the same memory address. Reference variables must be initialized when they are declared. Once created, a reference variable cannot be changed.

```cpp
// File: references5.cpp

#include <iostream>
using namespace std;

int main(){
    int x = 13;
    int &y = x; // y is an alias for x

    cout << "x = " << x
         << "    y = " << y << endl;

    // Changing y also changes x
    y = 22;
    cout << "x = " << x
         << "    y = " << y << endl;

    // Look at the memory address of x and y
    cout << "Address of x = " << &x << endl
         << "Address of y = " << &y << endl;
    return EXIT_SUCCESS;
}
```

Listing 6.12: references5.cpp

```
x = 13    y = 13
x = 22    y = 22
Address of x = 0020F95C
Address of y = 0020F95C
```

Figure 6.10: Output of references5.cpp

6.6 Function Templates

Function templates are used for *generic programming*. This is when a single function is written that has a special *type parameter*. The function is compiled multiple times, once for each different type that is actually used with that function. An example of using templates is given in Listings 6.14 and 6.13 and Figure 6.11

```
// File LargestTemplate.h
#ifndef LARGESTTEMPLATE_H
#define LARGESTTEMPLATE_H

template < class T >
T largest(T value1, T value2, T value3){
    T largestValue = value1;

    if(value2 > largestValue)
        largestValue = value2;

    if(value3 > largestValue)
        largestValue = value3;

    return largestValue;
}

#endif
```

Listing 6.13: LargestTemplate.h

```
// File: LargestTemplate.cpp

# include <iostream>
# include <string>
# include "LargestTemplate.h"

using namespace std;

int main() {
    int int1, int2, int3;
    string str1, str2, str3;

    cout << "Enter three integers: ";
    cin >> int1 >> int2 >> int3;
    cout << "The largest is " << largest(int1, int2, int3) << endl;

    cout << "\nEnter three strings: ";
    cin >> str1 >> str2 >> str3;
    cout << "The largest is " << largest(str1, str2, str3) << endl;


    return EXIT_SUCCESS;
}
```

Listing 6.14: LargestTemplate.cpp

```
Enter three integers: 10 30 15
The largest is 30

Enter three strings: Cat Dog Zebra
The largest is Zebra
```

Figure 6.11: Output of LargestTemplate.cpp

6.7 `cmath` Library Functions

The `cmath` declares a set of functions to compute common mathematical operations and transformations. These functions are listed in Figure 6.12.

`abs`	`int abs(int n)`	absolute value of integer parameter.
`labs`	`long abs(long n)`	absolute value of long integer parameter.
`fabs`	`double fabs(double n)`	absolute value of floating point parameter.
`sin`	`double sin(double a)`	sine of angle a (a given in radians).
`cos`	`double cos(double a)`	cosine of angle a (a given in radians).
`tan`	`double tan(double a)`	tangent of angle a (a given in radians).
`asin`	`double asin(double x)`	arc sine of x.
`acos`	`double acos(double x)`	arc cosine of x.
`atan`	`double atan(double x)`	arc tangent of x.
`atan2`	`double atan(double y, double x)`	arc tangent of y/x.
`sinh`	`double sinh(double a)`	hyperbolic sine of angle a (a given in radians).
`cosh`	`double cosh(double a)`	hyperbolic cosine of angle a (a given in radians).
`tanh`	`double tanh(double a)`	hyperbolic tangent of angle a (a given in radians).
`ceil`	`double ceil(double x)`	smallest integer that is greater or equal to x.
`floor`	`double floor(double x)`	largest integer that is less than or equal to x.
`fmod`	`double fmod(double x, double y)`	remainder of floating point division x/y.
`exp`	`double exp(double x)`	exponential value of parameter x, i.e. e^x.
`log`	`double log(double x)`	natural logarithm of x ($\log_e x$).
`log10`	`double log10(double x)`	logarithm base 10 of parameter x ($\log_{10} x$).
`pow`	`double pow(double x, double y)`	x raised to the power of y (x^y).
`sqrt`	`double sqrt(double x)`	square root of parameter x ($\sqrt{x}$).
`frexp`	`double frexp(double x, int *exp)`	Get mantissa and exponent of floating-point, $x = mantissa^{exponent}$. *mantissa* is returned, 2^{nd} parameter set to *exponent*.
`ldexp`	`double ldexp(double x, int exp)`	Get floating-point value from mantissa and exponent. Returns $x \times 2^e xp$.
`modf`	`double modf(double x, double *ipart)`	Split floating-point into fractional and integer parts. Breaks x into: the integer (stored in location of ipart) and the fraction (returned).
`atof`	`double atof(const char *string)`	Convert string to double.

Figure 6.12: Functions in the `cmath` library.

6.8 Other Standard C++ Header Files

Figure 6.13 lists many of the C++ standard header files that contain a variety of built in functions.

`<iostream>`	Stream I/O, including standard input and output
`<iomainp>`	Formatting input and output
`<cstdlib>`	Convert between numbers and test, random numbers, memory allocation, and others
`<ctime>`	Time and date
`<cctype>`	For working with `char` variables, convert case, test if a `char` is a digit, etc.
`<cstring>`	For working with C-style strings
`<typeinfo>`	Determining data types at runtime
`<ecxeption>`	Used for exception handling
`<stdexcept>`	Used for exception handling
`<memory>`	Allocating memory to STL containers
`<fstream>`	File I/O
`<string>`	String class
`<sstream>`	Input from strings in memory and output to strings in memory
`<functional>`	Used by Standard C++ algorithms
`<iterator>`	For accessing data in STL containers
`<algorithm>`	Algorithms to manipulate STL containers
`<cassert>`	For adding assertions of invarients
`<cfloat>`	Floating point size limits of the system
`<climits>`	Integral size limits of the system
`<cstdio>`	C-style input and output
`<local>`	Used for different norms, such as monetary formats or languages
`<limits>`	Numeric data type limits on each computer platform
`<utility>`	Catch all used by many STL libraries

Figure 6.13: Selected C++ standard library header files.