

Chapter 7 – Non-Void Functions & Scope

1. Functions Revisited

Functions are named units of code that perform a specific action. A number of functions have already been introduced, such as *cout.width(int)* and *strcpy(string,string)*. Functions are used by calling them by name. Functions that are called preceded by an *object name*, such as *cout.fill(char)*, are referred to as *member functions* because they are members of the class of the object. Functions that are called by the function name only are referred to as *free functions* and are functions that are not members of a class.

Programmers not only use functions, but spend a lot of time and energy creating them. Functions are often created for one program. Other functions, such as those found in *iostream*, *string* and *string.h*, are used over and over again in many programs.

Free functions are sometimes referred to as functions of convenience. They are created whenever spending the time to make a function improves the functionality or reliability of a program. For example, much of the C++ code in the free functions found in *string.h* could be written by any knowledgeable C++ programmer. Since they would be need to be re-written over and over again, it makes sense to write them once, save them in a file, and include the file into whatever program needs C-strings. (Note: *string.h* is, by definition, part of the standard C++ programming environment and is available with any C++ compiler.)

Free functions created for a specific program are generally prototyped above and written below or included above the function *main*. This is not always so, but it is simplest to do so.

2. Non-Void Free Functions

A *non-void free function* accepts data in *parameters* when called and returns information once the function's processing finishes.

All functions consist of a header and a body.

header
body

Function headers have three parts:

return-type function-name (argument-list)

where *return-type* is a *type* or *class* name, such as *int*, *float*, *char*, etc.

function-name is whatever the creator of the function chooses,
within the limits of naming in C++

argument-list contains the parameter-objects to receive information
during a call or invocation of the function.

Here is an example of a function header for a function that will return the area of a circle:

```
double AreaC(double radius)
```

The body of a function follows the function header and is code enclosed in a compound statement. This has the following form:

```
header
{
  ... statements ...
}
```

Everything in a compound statement is treated as being one thing. In effect, compound statements make many statements into a single statement.

In non-void functions, the very last statement in the function is the *return* statement. The return statement has this form:

```
return expression;
```

The value calculated in the expression of the *return* statement will be returned from the function when the function finishes. This is just what happens when a calculator is used to compute the value of function (such as square root). Here is an example of a function header and body with a *return* statement:

```
double AreaC(double radius)
{
  return 3.1416 * radius * radius;
}
```

To be used in a program, the *name* of the function followed by an *argument* would have to be placed in an *expression* of *type double*. For example:

```
cout << "The volume of the cylinder is " << AreaC(r) * length;
```

Here is an example of a program that makes use of the free function *areaC*.

```
#include <iostream>
using namespace std;

double AreaC(double radius) {
    return 3.1416 * radius * radius;
}

void main() {
    cout << AreaC(3.7) << endl;
    cout << AreaC(55.0) << endl;
    cout << AreaC(23) << endl;
    cout << AreaC(.56) << endl;
}
```

Functions written in this way can only be called from a function that is located below the function. This can be a serious problem if there exist functions that call other functions. It may not even be possible, or at least very inconvenient to line up functions in such an order. To get around this problem, functions are usually *prototyped* above function *main*, then written, or implemented after function *main*. In its simplest form, a function prototype is just the function header written above the function *main*. The example program above could be written with a prototype as follows:

```
#include <iostream>
using namespace std;

double AreaC(double radius);

void main() {
    cout << AreaC(3.7) << endl;
    cout << AreaC(55.0) << endl;
    cout << AreaC(23) << endl;
    cout << AreaC(.56) << endl;
}

double AreaC(double radius) {
    return 3.1416 * radius * radius;
}
```

When either version of this program is run, the output looks like this:

```
86.017
19006.7
3323.81
1.97041
```

Here is another example of a program that makes use of the free function *AreaC*.

```
#include <iostream>
using namespace std;

double AreaC(double radius);

void main() {
    double r;
    double length;
    cout << "Enter the radius of the cylinder: ";
    cin >> r;
    cout << "Enter the length of the cylinder: ";
    cin >> length;
    cout << "The volume of the cylinder is " << AreaC(r) * length << endl;
}

double AreaC(double radius) {
    return 2.0 * 3.1416 * radius * radius;
}
```

If the user enters a radius of 5.1 and a length of 18.0 when the program is run, the output will look like the following.

```
Enter the radius of the cylinder: 5.1
Enter the length of the cylinder: 18.0
The volume of the cylinder is 2941.67
```

Because prototyping functions means that the programmer does not have to worry about the order of the functions will hamper his coding and since prototyping keeps the function main conveniently close to the top the file, it is customary to prototype all functions.

Functions may have more than one parameter. A call to a function must have as many arguments as the function header has parameters. When there are multiple arguments, data is passed in order. The first argument in the call goes to the first parameter in the function header, the second argument in the call goes to the second parameter in the header, etc. The following program includes a function *VolumeC* that uses multiple arguments.

```
#include <iostream>
using namespace std;

double AreaC(double radius);
double VolumeC(double radius, double length);

void main() {
    double r;
    double length;
    cout << "Enter the radius of the cylinder: ";
    cin >> r;
    cout << "Enter the length of the cylinder: ";
    cin >> length;
    cout << "The volume of the cylinder is " << VolumeC(r, length)
        << endl;
}

double AreaC(double radius) {
    return 2.0 * 3.1416 * radius * radius;
}

double VolumeC(double radius, double length) {
    return AreaC(radius) * length;
}
```

Notice that the function *VolumeC* calls the function *AreaC* and uses the return of *AreaC* in its calculation. In this example program, if the user enters a radius of 5.1 and a length of 18.0 when the program is run, the output will look like the following.

```
Enter the radius of the cylinder: 5.1
Enter the length of the cylinder: 18.0
The volume of the cylinder is 2941.67
```

2. Scope and Constants

Scope refers to the rules in C++ that govern when an object can be used and when it cannot. When an object can be used, it is said to be *visible*. When an object cannot be used, it is said to be *not visible*.

The following program uses the function `int Cube(int)` to return the cube of an argument.

```
#include <iostream>
using namespace std;

int Cube(int i);

void main() {
    int c = Cube(3);
    cout << c << endl;
}

int Cube(int i) {
    int theCube = i * i * i;
    return theCube;
}
```

The variable object *c* receives the return value of the function `Cube(int)`. *c* is then used in a `cout` statement. As written, the program outputs the value 27.

In the program, there are 3 variables. In the function `main`, there is the variable *c*, which can only be used inside the function `main`. If we try to use the variable *c* inside the function `Cube`, the compiler will generate an error message. Inside the function `Cube`, there are the variables *i* and `theCube`. If we try to use either variable in the function `main`, the compiler will generate an error message.

Variables are said to be *local* to the function that they are declared in. Once declared in a function, variables cannot be used outside of that function. Variables declared in a function are visible only in that function.

Objects declared outside of any function are said to be *global* objects and are visible in all functions that follow the declarations. Global objects can create problems when creating large programs. By convention, global objects are restricted to *named constants*, which are objects that receive a value when declared, but that value cannot be changed or varied when the program is running. Named constants are declared in the following manner:

```
const type object = value;
```

The following program makes use of the constant object PI:

```
#include <iostream>
using namespace std;

const double PI = 3.1416;

double Area(double r);

void main() {
    cout << Area(5.5) << endl;
}

double Area(double r) {
    return 2.0 * PI * r * r;
}
```

When run, this program outputs the value *190.067*.

The scope of PI is global and covers the entire program. PI may be used in an expression anywhere in the program after its declaration, but it may not be used on the left side of an assignment statement. For example, the following assignment statement would generate a compiler error.

~~PI = 3.1;~~

Exercises

1. What is a free function?
2. What is a non-void function?
3. What is the purpose of the return-type in a non-void function header?
4. What is the purpose of the argument list in a function header?
5. What is the purpose of a return statement in a non-void function?
6. What is scope?
7. What is a global object?
8. What is a local object?
9. What is a constant?

10. Find the errors in the following program.

```
#include iostream>
using namespace std;

const int A 1;

int Sum(int b int c);

void main( )
    int d = 2, e = 3
    cout < Sum(d, b);
}

int Sum(int b, int c)
    A + b + e;
}
```

11. What is the output of this program?

```
#include <iostream>
using namespace std;

const int A = 4;

int Something(int b) ;

void main() {
    int c = 12;
    cout << A << " " << c << endl;
    c = Something(c);
    cout << A << " " << c << endl;
}

int Something(int b) {
    cout << A << " " << b << endl;
    b = A + 2;
    cout << A << " " << b << endl;
    return b;
}
```

Programming Exercise 7.1

Complete the program below by creating a non-void free function named *BoxV* (see the call of *BoxV* in function *main*) that receives the length, width, and depth measurements of a box and returns the volume of the box.

```
#include <iostream>
using namespace std;

... put your function prototype here ...

int main( ) {
    double l, w, d, v;
    cout << "Enter Length, Width, and Depth in feet: ";
    cin >> l >> w >> d;
    v = BoxV(l,w,d);
    cout << "The volume of the box is " << v << " feet";
    return 0;
}

... put your function here ...
```

Programming Exercise 7.2

Create and use a function that accepts an amount in U.S. Dollars as the value of an argument and returns the equivalent amount in French Francs. (The rate on 2-21-99 was one U.S. Dollar to 5.9229 French Francs. Current rates are usually available in newspapers and on the Internet at such sites as www.x-rates.com.) Create a program that uses this function to calculate and output the amount of French Francs for a user input amount of U.S. Dollars. All input and output must be in the *main* function.