Chapter 4

Functions

A *function* is a subprogram that is executed by being *called* from within another function. Every C++ program must include a function named `main()`. That is where program execution begins. A function is called by using its name as variables, as in

```
double y = sqrt(x);
```

or as an independent statement, as in

```
print(a, n);
```

Functions allows *program modularization* which is essential for good software development.

4.1 FUNCTION DECLARATIONS AND DEFINITIONS

Like a variable, a function must be declared before it is called. A *function declaration* has three parts: its return type, its name, and its parameter list. A function declaration should also include a comment that describes what the function does.

EXAMPLE 4.1 A Function Declaration

```
double power(double x, int n);   //PRECONDITION: x > 0
// Returns the value of x raised to the power n
```

Here, **`double`** is the return type, `power` is the name of the function, and `{double x, int n)` is its parameter list. This function has two parameters: `x` and `n`.

A function *parameter list* is a list of variable declarations enclosed in parentheses. The variables are called *parameters* (or *formal parameters).* The parameter list may be empty, but the enclosing parentheses are still required.

A function declaration is also called a *prototype.* It contains the minimal information that the compiler needs to compile code that uses the function. Consequently, the names of the parameters may be omitted, like this:

```
double power(double, int);
```

However, it is usually better to include the parameter names.

A *function definition* contains all the information needed to execute code that calls it: its header and its main block of executable statements.

EXAMPLE 4.2 A Function Definition

```
double power(double x, int n)
{   double y = 1.0;
    for(int i=0; i<n; i++) // if n>0, return x*x*...*x (n times)
        y *= x;
    for(int i=0; i>n; i—)       // if n<0, return 1/{x*x*...*x) (n times)
        y /= x;
    return y;
}
```

Note that only one of the two loops will execute (or neither if `n = 0`).

If the function definition is given before it is called, then it does not need a separate declaration.

EXAMPLE 4.3 Declaring a Function with Its Definition

Here is a complete C++ program:

```
#include <iostream>
using namespace std;
double power(double x, int n)
{   double y = 1.0;
    for(int i=0; i<n; i++) // if n > 0, return x*x*...*x (n times)
        y *= x;
    for(int i=0; i>n; i--) //if n < 0, return 1/(x*x*...*x) (n times)
        y /= x;
    return y;
```

```
}

int main()
{   cout << power(7.4, 3) << endl;
    cout << power(7.4, -3) << endl;
}
```

The output is

```
405.224
0.00246777
```

The alternative is to give only the function's declaration prior to the code from which it is called. In this case, the function definition is given elsewhere, either later in the file or in another file linked to the file that contains the code from which it is called.

EXAMPLE 4.4 Separating the Function Declaration from Its Definition

Here is another complete C++ program, equivalent to that in Example 4.3:

```
#include <iostream>
using namespace std;
double power(double x, int n);
// PRECONDITION: x > 0
// Returns the value of x raised to the power n

int main()
{   cout << power(7.4, 3) << endl;
    cout << power(7.4, -3) << endl;
}

double power(double x, int n)
{   double y = 1.0;
    for(int i=0; i<n; i++) // if n > 0, return x*x*...*x (n times)
        y *= x;
    for(int i=0; i>n; i--) // if n < 0, return 1/(x*x*...*x) (-n times)
        y /= x;
    return y;
}
```

The parameter names may be omitted from a function declaration (but not from its definition). For example, `double power (double, int);` is a valid declaration. The only purpose of the declaration is to provide the compiler with the information it needs to compile calls to the function.

The **`return`** statement in a function specifies the value to be returned to point where the function was called. In Example 4.4, the statement **`return`** `y;` terminates the execution of the function and returns the value of `y` (61043.7) to the **`cout`** statement in `main()` where the function was called.

Note that the `main()` function itself has a return statement. Its return type is **`int`**, so we usually simply return 0. The value returned can be used by the operating system after the program has terminated.

A function may have several **`return`** statements. Although it is usually best to place one at the end of the function definition, a **`return`** statement can be placed anywhere.

EXAMPLE 4.5 A Function with More than One return Statement

```
int factorial(int n)
// Returns n! = n*(n-1)*(n-1)*...*2*1
{   assert (n >= 0);
    if(n < 2) return 1; // 0! = 1 and 1! = 1
    int f = 1;
    while(n > 1)
        f *= n--;
    return f;
}
```

The condition `(n < 2)` in the **`if`** statement will be true only if the value of `n` is 0 or 1, and in both of those two special cases the factorial function should return 1. The function would be correct without the if statement, because in those two

special cases the `while` loop will be skipped and the return value of `f` is `l`. But the separate **`return`** statement within the **`if`** statement makes the function more efficient, because in the two special (and not uncommon) cases it avoids the declaration of the local variable `f` and the evaluation of the condition `(n > 1)`.

4.2 `void` FUNCTIONS

A **`void`** function is a function whose return type is **`void`**. This means that it returns no value. lt may have one or more **`return`** statements, but they must have the simpler form: **`return`**;

EXAMPLE 4.6 A `void` Function

This **`void`** function has no **`return`** statements:

```
void printLiteralDigit(int n)
// Prints the digit n in literal form.
// EXAMPLE: printLiteralDigit(7) would print "seven".
// PRECONDITION: n >= 0 && n <= 9
{   assert(n >= 0 && n <= 9);
    if(n == 0) cout << "zero";
    else if(n == 1) cout << "one";
    else if(n == 2) cout << "two";
    else if(n == 3) cout << "three";
    else if(n == 4) cout << "four";
    else if(n == 5) cout << "five";
    else if(n == 6) cout << "six";
    else if(n == 7) cout << "seven";
    else if(n == 8) cout << "eight";
    else cout << "nine";
}
```

A void function is called by using its name as an executable statement. The function that is defined in Example 4.6 could be called like this:

```
int main()
{    printLiteralDigit(7);
}
```

Since the function name is used like an independent C++ statement, it is usually helpful to use a verb phrase for the name of a **`void`** function.

4.3 TRACING A FUNCTION

It is wise to *trace* a function to ensure that it works properly. This means to track through the execution of calls to the function, executing its statements by hand (or with the aid of a calculator) to check the correctness of its logic. Each value of each variable should be shown.

EXAMPLE 4.7 Tracing a Function

Here is a trace of the call `power(4.0, 5)`:

x	n	i	y
4.0	5		1.0
		0	4.0
		1	16.0
		2	64.0
		3	256.0
		4	1024.0
		5	

The function returns 1024.0, which is the correct value for 4.0^5.

Here is a trace of the call `power(4.0,-3)`:

x	n	i	y
4.0	3		1.0
		0	0.25
		1	0.0625
		2	0.015625
		3	

The function returns 0.015625, which is the correct value for $4.0^{-3} = 1/64$.

Here is a trace of the call `power(4.0,0)`:

x	n	i	y
4.0	3		1.0
		0	

The function returns 1.0, which is the correct value for any positive number raised to the power 0.

4.4 TEST DRIVERS

A *test driver* is a complete program whose sole purpose is to test a function. It should be short, simple, obvious, and easy to use.

EXAMPLE 4.8 A Test Driver for the `power()` Function

```
main()
{   double x;
    int n = 1;
    while(n != 0)
    {   cin >> x >> n;
        cout << "power(" << x << "," << n << ") = " << power(x, n) << endl;
    }
    return 0;
}
```

Note how Spartan this program is. It has no documentation (comments) or user prompts. Those features are important in ordinary (long-lived) programs. But test drivers are only temporary programs. They are used only by the function's creator and only long enough to test the function. A thorough testing with this driver might look like this:

```
4.0 5
power(4,5)= 1024
4.0 -3
power(4,-3)= 0.015625
10 100
power(10,100)= le+100
100 10
power(100,10)= le+20
100 -10
power(100,-10)= le-20
10 -100
power(10,-100)= le-100
1 100
power(1,100)= 1
1 -100
power(1,-100)= 1
100 1
power(100,1)= 100
100 -1
power(100,-1)= 0.01
12345 1
power(12345, 1)= 12345
12345 0
power(12345, 0)= 1
```

The best strategy is to choose input values whose output is predictable. For example, the call `power(13579.08642, 28)` is just as good as `power(100, 10)` for checking the function's logic; but it is a bad choice because its output is not predictable (even with a good calculator!).

It is also important to try to test all the different cases and "boundary conditions" of a function. For example, the three obvious cases `n > 0`, `n == 0`, and `n < 0` for the `power()` function should be tested. Also, the special case `x == 1` should be tested.

Testing a function is a difficult art. You can never prove that a (non-trivial) function is correct just by testing it. But extensive testing is usually the best way to discover logical errors before the function is used in software development.

4.5 USING THE `assert()` FUNCTION TO CHECK PRECONDITIONS

Most function do not work properly on all possible values of their parameters. Restrictions on these parameter values are called *preconditions.* These preconditions should be listed clearly in comments that accompany the function's declaration. But those comments won't help prevent improper values from being passed to the function.

One simple but effective method for handling illegal parameter values is to use the standard `assert()` function to check preconditions. This function is defined in the `<assert>` header file. The function is used by passing the precondition to it as a `bool` expression. When the function is called, it evaluates the expression. If it evaluates to `false`, then the function aborts the program and prints a message reporting that the assertion failed.

EXAMPLE 4.9 Using `assert()` to Check Preconditions

```
#include <assert>

double power(double x, int n)
{   assort(x > 0);
    double y = 1.0;
    for(int i=0; i<n; i++) //if n > 0, return x*x*.. . *x (n times)
        y *= x;
    for(int i=0; i>n; i--) //if n < 0, return 1/(x*x*...*x)(-n times)
        y /= x;
    return y;
}
```

Here is what happens on a UNIX system when the precondition is violated:

```
-4.0 5
ex0407.cc:10: failed assertion 'x > 0'
IOT trap
```

The value `-4.0` was input for `x`. The first line of output reports that the assertion failed at line 10 in the program whose source code is in the file `ex0407.cc`. The second line simply classifies the abortion as an "IOT trap".

Technically, `assert` is a *parametrized macro.* Unlike a real function that gets compiled, a macro works more like an `include` directive. The compiler actually replaces the expression that uses its name with other code defined in the `<assert>` file, and then that code gets compiled. This replacement process is called *expanding* the macro.

4.6 PREDICATES

A *predicate* is a boolean function; *i.e.*, a function whose return type is `bool`. It is used to test some condition about its arguments.

EXAMPLE 4.10 The Predicate `is_prime()` Function

This function tests the condition that its argument is a prime number:

```
bool is_prime(int n)
// Returns true iff n has no divisors except 1 and itself
{   assert(n > 0);
    if(n == 1) return false;
    if(n == 2) return true;      // 2 is the first prime
    if(n % 2 == 0) return false; // 2 is the only even prime
    for(int d=3; d<n; d += 2)    // look for an odd divisor
        if(n % d == 0) return false;
    return true;    // no odd divisors were found
```

Here is a test driver for the `is_prime()` function:

```
int main()
{   int n = 2;
    while(n != 1)
    {   cin >> n;
        if(is_prime(n)) cout << n << " is_prime\n";
        else cout << n << " is not prime\n";
    }
    return 0;
}
```

The symbol “iff” stands for "if and only if." We use it to describe boolean functions. For example, here it means that the function returns `true` if `n` has no nontrivial divisors *and* it returns **`false`** if `n` *does* have some nontrivial divisor.

Comments are often essential for the understanding of a block of code. But the programmer should strive to make his or her code *self-documenting.* This means to choose programming structures that clarify the underlying logic and to choose names that describe what the things being named represent. In the case of function names, it is usually best to use only noun phrases for non-void functions, use only verb phrases for void functions, and use only predicate phrases for boolean functions:

```
double power(double x, int n);  // "power" is a noun
void print_literal_digit(int n);    // "print" is a verb
bool is_prime(int n);           // "is prime" is a predicate
```

4.7 DEFAULT ARGUMENTS

The expressions listed between the parentheses in a function call are called *arguments* (or *actual parameters*). For example, `4` and `5` are the arguments in the call `power(4, 5)`.

It is possible to define default values for some or all of the arguments simply by specifying those values as initializations in the function's parameter list. In computer science, the word *default* means a value that is used by the system when a specific value is not given. So a *default argument* is a value that is assigned to a parameter in place of a missing argument in a function call.

EXAMPLE 4.11 Specifying Default Arguments

```
double power(double x=1.0, int n=2)
{   assert(x > 0);
    double y = 1.0;
    for(int i=0; i<n; i++) //if n > 0, return x*x*...*x (n times)
        y *= x;
    for(int i=0; i>n; i--) // if n < 0, return 1/(x*x*...*x) (-n times)
        y /= x;
    return y;
}
```

Here, the parameter `x` is given the default argument `1.0`, and the parameter `n` is given the default argument 2. Then the function could be called like this:

```
cout << power(7.4, 3) << endl;
cout << power(7.4) << endl;
cout << power() << endl;
```

The output from these three calls would be

```
405.224
54.76
1
```

These are the correct values for 7.4^3, 7.4^2, and 1.0^2.

Note that all, some, or none of the default arguments may be used in a function call. If fewer arguments than parameters are passed, then the system matches the given arguments with the parameters one-to-one, scanning left-to-right. When it runs out of arguments, it uses the default values for the remaining parameters.

A function may define default arguments for all, some, or none of its parameters. The only rule is that, to be consistent with the process described above, the parameters with default values must follow those without default values in the parameter list.

4.8 PASSING BY `const` VALUE, BY REFERENCE, AND `const` REFERENCE

There are four different ways that an argument can be passed to a parameter in a function call: by value, by **`const`** value, by reference, and by **`const`** reference. These distinct methods are determined by the function's parameter list, by

preceding the parameters type with the **const** keyword and/or by preceding the parameter's name with the reference symbol &:

```
void fl(int x );     // x  is a value parameter
void f2(const int x ); // x is a const value parameter
void f3(int& x);    //  x  is a reference parameter
void f4(const int& x);     // x is a const reference parameter
```

Passing an argument by value is the simplest and most common method. In this case, the parameter is a separate local variable that exists only during the execution of the function. The call `f1(u);` to the first function declared will copy the value of the argument `u` into the parameter `x.` Then, any changes made to `x` during the execution of the function will have no effect upon the variable `u.` Note that since this method only uses the value of the argument, it could be a constant or even a general expression. For example, `f1(44)` and `fl(2*u - 3*v)` would be valid calls.

Passing an argument by **const** value is the same as passing it by value except that the **const** keyword makes the parameter a constant, prohibiting the function from changing its value. This is the kind of restriction that good programmers often force upon themselves to prevent coding errors. If you don't want your function to change its parameter value, then make it a constant. That tells the compiler to "keep you honest." It will alert you if you accidentally code a change.

EXAMPLE 4.12 Passing by const Value

```
void f2(const int x)   // x  is a const value parameter
{   cout << "The value of the const parameter x is " << x << endl;
}

int main()
{   f2(44);      // passing a constant
    int u = 55;
    f2(u);      // passing a variable
    int v = 33;
    f2(2*u - 3*v); // passing an expression
    return 0;
}
```

The output is

```
The value of the const parameter x is 44
The value of the const parameter x is 55
The value of the const parameter x is 11
```

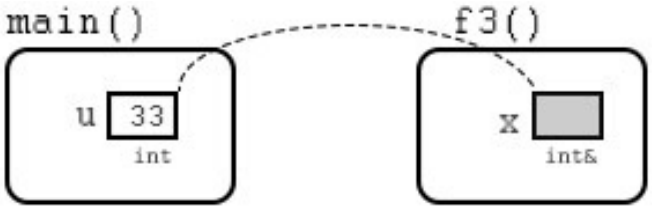
Passing an argument *by reference* allows the function to change the value of the argument. That is because the function's parameter is only a synonym (another name) for the existing argument.

EXAMPLE 4.13 Passing by Reference

```
void f3(int& x)  // x is a reference parameter
{   x *= 3;
}
```

The function triples the value of `x`. But `x` is just a synonym or "alias" for the argument passed to it, so the effect of the function is to triple the value of its argument.

```
int main()
{   int u = 33;
    f3(u);
    cout << u << endl;
    return 0;
}
```



The output from this program is

```
99
```

The diagram above may help to explain the relationship between the argument `u` and the parameter `x`: The parameter `x` does not have its own storage area; it is just another name for the variable `u`. So when the statement `x *= 3` executes, it triples the value of `u`.

Note that an argument that is passed by reference must be an actual variable. It is not possible to pass a constant or an expression by reference. That should seem reasonable because constants and expressions are read-only; values cannot be assigned to them. So only the second of the three calls made to `f2()` in Example 4.12 would work with `f3()`.

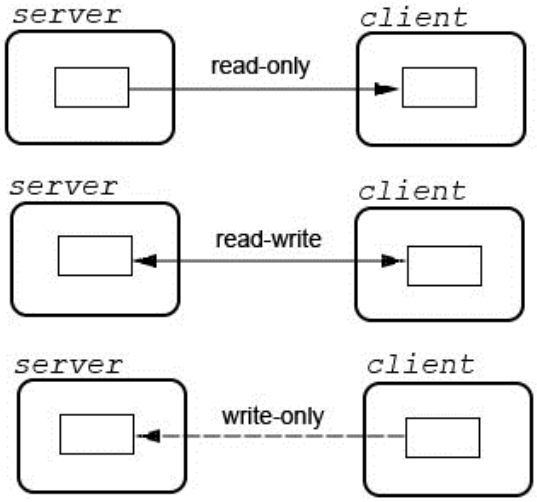
Passing an argument *by **constant** reference* is the same as passing it by reference except that the function is prohibited from changing its value. This may seem like a contradiction since the purpose of passing an argument by reference is often to change its value. However, there are other situations when the purpose is simply to avoid duplicating the argument (which is what happens when it is passed by value). That is the situation when the argument is a large object.

EXAMPLE 4.14 Passing by `const` Reference

```
void f4(const int& x} // x is a const reference parameter
{   cout << x << endl;
}
```

The reference here is not essential for the function to work properly.

When one system accesses a storage area of another system, there are three general modes of access: read-only, write-only, and read-write. The accessor is called the *client* and the provider is called the *server*. For example, if we think of the computer's processor as a client and its input-output devices as servers, then the keyboard is a read-only device, the hard disk is a read-write device, and the printer is a write-only device. We can also think of the `main()` function as a server and a function that it calls as its client. In that context, we can see that a function's value parameters are read-only, its function return value is write-only, and its reference parameters are read-write. In other programming languages (namely, Ada), these are called *in parameters*, *out parameters*, and *in-out parameters.*



4.9 RETURNING BY REFERENCE

Non-`void` functions usually return by value. Like passing by value, this means that a copy of an existing variable or expression is returned. But like arguments, returns can also be made by reference. The advantage is the same: it avoids duplicating an object.

EXAMPLE 4.15 Returning a Reference

This function triples `x` and then returns it by reference:

```
int& f(int& x)
{   cout << "x = " << x << endl;
    x *= 3;
    cout << "x = " << x << endl;
    return x;
```

```
}
```

Note that the local variable `x` is a synonym for whatever argument is passed to the function.

Here is a test driver:

```
int main()
{   int m = 11;
    cout << "m = " << m << endl;
    int  n = f(m);
    cout << "m = " << m << endl;
    cout << "n = " << n << endl;
    f(n) = 44;
    cout >> "n = “ << n << endl;
}
```

The output is

```
m = 11
x = 11
x = 33
m = 33
n = 33
x = 33
x = 99
n = 44
```

The first call `f(m)` has the effect of tripling the value of `m` from `11` to `33`. That value is then assigned to `n`. The second call `f(n)` is the interesting one because that expression is placed on the left side of an assignment: it looks like the value 44 is being assigned to the function call. Actually, 44 is being assigned to the reference that the function returns. That is a reference to the local variable `x`, which itself is a reference to the argument `n`. So the end result of the assignment `f(n) = 44` is to change the value of `n` to 44. Note that, prior to this final assignment, `x` (and therefore `n`) was changed from `33` to 99. But that value is then replaced by 44 in the final assignment `f(n) = 44`.

Note that if a function returns **an** object by reference, it must be an object that existed before the function was called because it has to exist after the function has returned.

In C++, an *object* is a contiguous region of memory. The simplest objects are variables and constants. An *lvalue* is an expression that refers to an object. The simplest lvalues are names of variables and constants. The expression `f(n)` in Example `4.15` is an lvalue. It refers to the object named `n`. The term "lvalue" originally meant "anything that can be on the left side of an assignment." But that definition no longer applies because an lvalue can refer to a constant. An lvalue that can be on the left side of an assignment is now called a *modifiable lvalue.*

4.10 OVERLOADING A FUNCTION NAME

C++ allows different functions to have the same name. This is called *overloading.* The only requirement is that they have different parameter lists; *i.e.,* either a different number of parameters or different types in at least one parameter slot.

EXAMPLE 4.16 Overloading the `swap()` Function

Here are two different functions, both named `swap ()`:

```
void swap(int& x, int& y)
{   int temp = x;
    x = y;
    y = temp;
}
void swap(float& x, float& y)
{   float temp = x;
    x = y;
    y = temp;
}
```

Here is a test driver:

```
int main()
{   int m=22, n=44;
    cout << "m = " << m << ", n = " << n << endl;
    swap(m, n);
    cout << "m = " << m << ", n = " << n << endl;
    float s=2.2, t=4.4;
    cout << "s = " << s ", t = " << t << endl;
    swap(s, t);
    cout << "s = " << s ", t = " << t << endl;
}
```

Here is the output:

```
m = 22, n = 44
m = 44, n = 22
s = 2.2, t = 4.4
s = 4.4, t = 2.2
```

The purpose of each function is simply to interchange the values of the two arguments passed (by reference). But the types of the arguments must match the types of the parameters, so being able both to swap integers and to swap floats requires two separate functions. Overloading simply allows us to use the same name `swap()` for both of them.

Note that the fact that these two functions perform the same operations is irrelevant to the issue of overloading, as are the facts that they are `void` functions and that they use reference parameters.

EXAMPLE 4.17 Overloading Different Functions

Here are three very different functions that overload the name `f()`:

```
void f(char c)
{   c = (c >= 'a' && c<= 'z' ? c - 'a' + 'A' : c);
    cout.put(c);
}
int f(int n)
{   return (n%2 == 0 ? n/2 : 3*n + 1);
}
float f(float x, float y, float z)
{   y = (y > x ? y : x);
    return (z > y ? z : y);
}
```

This overloading doesn't make much sense. It simply illustrates how different overloaded functions can be.

Review Questions

4.1 Does a function have to be declared before it is called?

4.2 What is the difference between a function declaration and a function definition?

4.3 What is a function prototype?

4.4 How many return statements can a function have?

4.5 Does a function have to return a value?

4.6 What is the difference between a value parameter and a reference parameter?

4.7 What is the difference between a `const` parameter and a non-`const` parameter?

4.8 What is the difference between returning by value and returning by reference?

4.9 What is wrong with the following version of the `swap()` function:

```
void swap(int x, int y)
{   int temp = x;
    x = y;
    y = temp;
```

```
}
```

4.10 What is wrong with the following version of the `swap()` function:

```
void swap(int& x, int& y)
{   x = y;
    y = x;
}
```

Problems

4.11 Trace the call `power(2, 5)` to the function defined in Example 4.2.

4.12 Trace the call `factorial(5)` to the function defined in Example 4.5.

4.13 Trace the call `is prime (121)` to the function defined in Example 4.10.

4.14 Trace the call `swap (m, n)` to the first function defined in Example 4.16 on page 77, assuming that the original values of `m` and `n` are 44 and 88, respectively.

Programming Problems

4.15 Implement the following function and test it with a test driver:

```
float mean(float x, float y, float z);
// Returns the mean average of x, y, and z.
// EXAMPLE: mean(4, 7, 4) returns 5.0;
```

4.16 Implement the following function and test it with a test driver:

```
float min(float x, float y, float z);
// Returns the smallest of x, y, and z.
// example: min(4, 7, 4) returns 4.0;
```

4.17 Implement the following function and test it with a test driver:

```
float median(float x, float y, float z);
// Returns the middle number among x, y, and z.
// EXAMPLE: median{4, 7, 4) returns 4.0;
```

4.18 Implement the Babylonian Algorithm (Algorithm 1.1 on page 1) as modified in Problem 1.20:

```
double sqrt(double x);
// Returns the square root of x.
// PRECONDITION: x >= 0.
// EXAMPLE: sqrt(49.0) returns 7.0;
```

4.19 Implement Algorithm 1.2 on page 5 to convert binary numerals to decimal:

```
int decimal(int b);
// Returns the decimal numeral whose value equals that
// represented in the binary form b.
// PRECONDITION: each digit of b is a bit: 0 or 1
// EXAMPLE: decimal(10001011) returns 139
```

4.20 Implement Algorithm 1.4 to convert decimal numerals to binary:

```
int binary(int n);
// Returns the binary numeral that represents n.
// PRECONDITION: n >= 0
// POSTCONDITION: each digit of the integer returned is a bit
// EXAMPLE: binary(139) returns 10001011
```

4.21 Implement the following function which uses Horner's method (see Problem 1.34) to evaluate the polynomial $2x^5 - 7x^4 + 6x^3 + 9x^2 + 8x - 5$:

```
double p(double x);
// Returns 2x^5 - 7x^4 + 6x^3 + 9x^2 + 8x - 5.
// EXAMPLE: p(4.1) returns 931.70732
```

4.22 Using only subtraction, implement the following function which returns the remainder from integer division:

```
int mod(int n, int d);
// Returns n%d.
// PRECONDITION: n >= d > 0
// POSTCONDITION: (n/d)*d + r == n, where r is returned
// EXAMPLE: mod(44, 7) returns 2
```

4.23 Implement the quadratic formula. (See Problem 3.11):

```
int solutions(float& xl, float& x2, float a, float b, float c);
// Modifies xl and x2 so that they are the solutions to the
```

```
// equation a*x*x + b*x + c == 0, and then returns either 0, 1,
// 2, or 3, according to whether the number of distinct
// solutions is 0, 1, 2, or infinite.
// POSTCONDITION: if 1 is returned, then xl is the unique sol'n;
// if 2 is returned, then xl and x2 are the distinct sol'ns
// EXAMPLE: solutions(xl, x2, 1, 6, 9) returns 1 with xl == -3.0
```

4.24 Implement the solution to Problem 2.18 as the function:

```
float centimeters(float x);
// Returns the number of centimeters in x inches.
// PRECONDITION: x > 0
// POSTCONDITION: the returned value y == 2.54*x
// EXAMPLE: centimeters(200) returns 508.0
```

4.25 Implement the solution to Problem 2.19 as the function:

```
float celsius(float x);
// Returns the temperature in Celsius degrees for a given
// temperature in Farenheit degrees.
// PRECONDITION: x >= -273.0
// POSTCONDITION: the returned value y == 5*(x - 32)/9
// EXAMPLE: celsius(68) returns 20.0
```

4.26 Implement the solution to Problem 2.21 as the function:

```
void convert(int& weeks, int& days, int& hours, int x);
// Modifies weeks, days, and hours so that they represent the
// same time duration as x hours.
// PRECONDITION: x >= 0
// POSTCONDITION: (7+weeks + days)*24 + hours == x
// EXAMPLE: convert(w, d, h, 4000) makes w = 23, d = 5, h = 16
```

4.27 Implement the solution to Problem 2.23 as the function:

```
void reverse(int& n) ;
// Reverses the digits of n.
// PRECONDITION: 100,000 <= n <= 999,999
// EXAMPLE: if n == 289405, reverse(n) makes n == 504982
```

4.28 Implement Example 3.20 on page 50 as the function:

```
void print(int n) ;
// Prints the sequence of numbers n, nl, n2, ..., 1, where the
// successor of each n > 1 is either n/2 or 3*n + 1 depending on
// whether n is even or odd.
// PRECONDITION: n > 0
// EXAMPLE: print(3) prints 3, 10, 5, 16, 8, 4, 2, 1
```

4.29 Implement Example 3.27 as the function:

```
double power(double x, int n);
// Returns x^n; i.e., x raised to the nth power.
// PRECONDITION: x > 0
// EXAMPLE: power(2.01, 3) return 8.120601
```

4.30 Implement the solution to Problem 3.12 on page 57 as the function:

```
void print_date(int n);
// Prints the month and day of the month for the year day n,
// assuming that the year is not a leap year.
// PRECONDITION: 1 <= n <= 365
// EXAMPLE: print_date(65) prints: March 6
```

4.31 Implement the integer binary logarithm function. (See Problem 3.15.):

```
int lg(int n);
// Returns the number of times n can be divided by 2.
// PRECONDITION: n > 0
// POSTCONDITION: 2^p <= n < 2^(p+l), where p is returned
// EXAMPLE: lg(100) returns 6
```

4.32 Implement the Euclidean Algorithm (Algorithm 3.1 on page 59) as the function:

```
int gcd(int m, int n);
// Returns the greatest common divisor of m and n.
// PRECONDITIONS: m > 0, n > 0
//POSTCONDITIONS: g is a factor of both m and n, and no x > g
// is a factor of both m and n
// EXAMPLE: gcd(252, 441) returns 63
```

4.33 Implement the solution to Problem 3.19 as the function:

```
int sum(int n);
```

```
// Returns the sum of the first n positive integers.
// PRECONDITION: n > 0
// EXAMPLE: sum(10) returns 55
```

4.34 Implement the solution to Problem 3.20 as the function:

```
int sum(int n);
// Returns the sum of the first n squares.
// PRECONDITION: n > 0
// EXAMPLE: sum(10) returns 385
```

4.35 Implement the solution to Problem 3.20 as the function:

```
int sum(int n);
// Returns the sum of the first n cubes.
// PRECONDITION: n > 0
// EXAMPLE: sum(10) returns 3025
```

4.36 Implement the solution to Problem 3.20 as the function:

```
int fact(int n);
// Returns n factorial: n! = 1*2*3*...*n.
// PRECONDITION: 0 <= n <= 12 (to avoid integer overflow)
// EXAMPLE: fact(5) returns 120
```

4.37 Implement the solution to Problem 3.20 as the function:

```
int perm(int n, int k);
// Returns the number of permutations of size k from a set of
// n elements.
// PRECONDITION: 0 <= k <= n <= 12
// EXAMPLE: perm(ll, 4) returns 7920
```

4.38 Implement the solution to Problem 3.24 as the function:

```
int comb(int n, int k);
// Returns the number of combinations (subsets) of size k from
// a set of n elements.
// PRECONDITION: 0 <= k <= n <= 33 // EXAMPLE: perm(ll, 4) returns 330
```

4.39 Implement the Fibonacci function. (See Problem 3.28):

```
int fib(int n);
// Returns the nth Fibonacci number.
// PRECONDITION: 0 <= n <= 4 6
// EXAMPLE: fib(10) returns 55
```

4.40 Implement the solution to Problem 3.31 as the function:

```
void amortize(float a, float r, float p);
// Prints the amortization schedule for a loan of an amount a,
// with an interest rate r, and a monthly payment of p.
// PRECONDITIONS: 0 < r < 1.0, 0.0 < p <= a
```

4.41 Implement the function:

```
bool is_prime_divisor(int d, int n) ;
// Returns true iff d is a prime divisor of n.
// PRECONDITION: 2 <= d <= n
// EXAMPLE: is_prime_divisor(7, 350) returns true because 7 is a
// prime number and a divisor of 350
```

Hint: use the `is_prime()` function from Example 4.10.

4.42 Implement the function:

```
void print_dollars(float x) ;
// Prints x in literal form as a dollar amount.
// PRECONDITION: x >= 0.00 && x < 1000.00
// EXAMPLE: print dollars(123.45) would print
// one hundred twenty-three dollars and 45 cents
```

Hint: use the `print literal digit` () function from Example 4.6

Solutions

4.1 Yes, a function must be declared before it is called. However, its complete definition also serves as its declaration, so if it is defined before it is called, then it does not need a separate declaration.

4.2 A function declaration consists of only its header: its return type, its name, and its parameter list. A function definition contains all the information about the function: its header and its body block of executable statement.

4.3 A function prototype is its declaration. It may omit the names of its parameters.

4.4 Anon-`void` function must have at least one `return` statement and may have more. The required `main` () function (whose return type is `int`) is an exception: Standard C++ does not require it to have a `return` statement. A `void` function does not have to have a `return` statement, although it may have several.

4.5 Each `return` statement in a non-`void` function must return a value of the same type as the function's declared return type. Each `return` statement in a `void` function must not return a value.

4.6 A value parameter is a copy of the argument passed to it. A reference parameter is simply another name for the argument passed to it. So an argument passed to a reference parameter must be a named object, whereas an argument passed to a value parameter may be a literal or an expression.

4.7 A `const` parameter is constant: the function cannot change its value. It can change the value of a non-`const` parameter.

4.8 When a function returns by value (the most common method), it sends a copy of the expression returned to the point in the previous function where it was called. When a function returns by reference, the function call expression itself becomes a synonym for the object to which it refers. That object must exist after the function returns, so it cannot be local to the returning function.

4.9 The parameters x and `y` are passed by value, so the function will have no effect upon the arguments passed to it.

4.10 It takes three steps to interchange two values. This two-step version assigns the value of `y` to x without first saving the value of x first, so that the original x-value is lost.

4.11 Trace of the call `power (2, 5):`

i	y
	1.0
0	2.0
1	4.0
2	8.0
3	16.0
4	32.0
5	

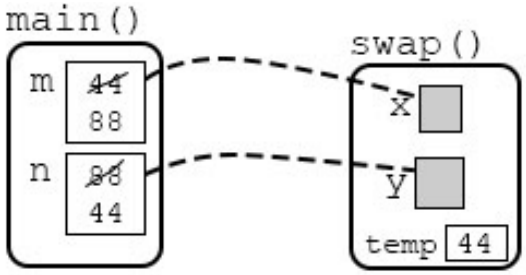
4.12 Trace of the call `factorial(5);`

f	N
1	5
5	4
20	3
60	2
120	1

4.13 Trace of the call `is_prime(121);`

d	n%d
3	1
5	14
7	2
960	4
11	0

4.14 Trace of the call `swap(m,n)`:



4.15
```
float mean(float x, float y, float z)
{   return (x + y + z) /3;
}
```
4.16
```
float min(float x, float y, float z)
{   if (x < y) y = x;
    if (y < z) z = y;
    return z;
}
```
4.17
```
float median(float x, float y, float z)
{   if (x <= y && y <= z || z <= y && y <= x) return y;
    if (x <= z && z <= y || y <= z && z <= x) return x;
    return x;
}
```
4.18
```
double sqrt(double x)
{   assert(x >= 0);
    double y = 1.0, z, r;
    do
    {   z = x/y;
        y = (y + z)/2;
        r = (y – z)/y;              // signed relative error
        r = (r < 0 ? –r : r);       // unsigned relative error
    } while(r > 5e-13);             // for 12-digit precision
    return y;
}
```
4.19
```
int decimal(int b)
{   int b0 – b%10;  assert(b0 == 0 || b0 == 1); b/= 10;
    int b1 – b%10;  assert(b1 == 0 || b1 == 1); b/= 10;
    int b2 – b%10;  assert(b2 == 0 || b2 == 1); b/= 10;
    int b3 – b%10;  assert(b3 == 0 || b3 == 1); b/= 10;
    int b4 – b%10;  assert(b4 == 0 || b4 == 1); b/= 10;
    int b5 – b%10;  assert(b5 == 0 || b5 == 1); b/= 10;
    int b6 – b%10;  assert(b6 == 0 || b6 == 1); b/= 10;
    int b7 – b%10;  assert(b7 == 0 || b7 == 1); b/= 10;
    int b8 – b%10;  assert(b8 == 0 || b8 == 1); b/= 10;
    int b9 – b%10;  assert(b9 == 0 || b9 == 1); b/= 10;
    return (((((((b9*2 + b8)*2 + b7)*2 +b6)*2 + b5)*2 + b4)*2
                        + b3)*2 + b2)*2 +b1)*2 + b0;
}
```
4.20
```
int binary(int n)
{   // same as Problem 4.21 except use 10 for 2 and 2 for 10
}
```
4.21
```
double p(double x)
{   return ((((2*x – 7)*x + 6)*x + 9)*x + 8)*x – 5;
}
```
4.22
```
int mod(int n, int d)
{   assert(n >= d && d > 0);
    while(n > d)
        n -= d;
    return n;
}
```
4.23
```
int solutions(float& xl, float& x2, float a, float b, float c)
{   if(a == 0)
```

```
    {   if(b == 0)
            if(c == 0) return 3;
            else return 0;
        xl =-c/b;
        return 1;
    }
    double d = b*b - 4*a*c;
    if(d < 0) return 0;
    if(d == 0)
    {   xl = -b/(2*a);
        return 1;
    }
    double sqrtd = sqrt(d);
    xl = (-b + sqrtd)/(2*a);
    x2 = (-b - sqrtd)/(2*a);
    return 2;
}
```

4.24

```
float centimeters(float x)
{   assert(x > 0);
    return 2.54*x;
}
```

4.25

```
float celsius(float x)
{   assert(x >= -273.0);
    return 5.0*(x - 32.0)/9.0;
}
```

4.26

```
void convert(int& weeks, int& days, int& hours, int x)
{   assert(x >= 0);
    hours = x%24;
    x /= 24;
    days = x%7;
    weeks = x/7;
}
```

4.27

```
void reverse (int& n)
{   assert(n >= 100000 && n <= 999999);
    int temp = n;
    n = temp%10;
    temp /= 10;
    n = 10*n + temp%10;
    temp /= 10;
    n = 10*n + temp%10;
    temp /= 10;
    n = 10*n + temp%10;
    temp /= 10;
    n = 10*n + temp%10;
    temp /= 10;
    n = 10*n + temp;
}
```

4.28

```
void print(int n)
{   assert(n > 0);
    while (n > 1)
    {   cout << n << ", ";
        if(n%2 == 0) n /= 2;
        else n = 3*n + 1;
    }
    cout << 1 << endl;
}
```

4.29

```
double power(double x, int n)
{   assert(x > 0);
    double y = 1.0;
    if(n < 0)
    {   x = 1.0/x;
        n = -n;
    }
    while(n > 0)
        if(n%2 == 0)  // i is even
        {   x * = x ;  // square z
```

```
            n /= 2;   // halve n
        }
        else      // i is odd
        {   y *= x;
            --n;
        }
    return y;
}
```

4.30
```
void print_date (int day
{   // same as the solution to Problem 3.12
}
```

4.31
```
int lg(int n)
{   assert(n > 0);
    int count = 0;
    while(n > 1)
    {   n /= 2;
        ++count;
    }
    return count;
}
```

4.32
```
int gcd(int m, int n)
{   assert( m  >  0  &&  n > 0);
    do
    {   while (m <= n)
            n -= m;
        int tmp = m;
        m = n;
        n = tmp;
    } while(m > 0);
    return n;
}
```

4.33
```
int sum(int n)
{   assert(n > 0);
    int s = 0;
    for(int i=l; i <= n; i++)
        s += i;
    return s;
}
```

4.34
```
int sum(int n)
{   // same as the solution to Problem 4.33 except use s += i*i;
}
```

4.35
```
int sum(int n)
{   // same as the solution to Problem 4.33 except use s += i*i*i;
}
```

4.36
```
int fact(int n)
{   assert(n >= 0 && n <= 12);
    int y = 1;
    for(int i=2; y *= i; i++)
        y *= i;
    return y;
}
```

4.37
```
int perm(int n)
{   assert(0 <= k && k <= n && n <= 12);
    int y = 1;
    for(int i=l; I <= k; i++, n--)
        y *= n;
    return y;
}
```

4.38
```
int comb(int n)
{   assert(0 <= k && k <=n && n <= 33);
    int y = 1;
    for(int i=l; I <= k; i++, n--)
        y = y*n/i;
    return y;
}
```

4.39
```
int fib(int n)
{   assert(n >= 0 && n <= 46);
    if(n == 0) return 0;
    int f0=0, fl=l, f2;
    for(int i=2; i <= n; i++)
    {   f2 = fl + fO;
        fO = fl;
```

```
            fl = f2;
        }
        return f1;
    }
```

4.40

```
    void amortize(float a, float r, float p)
    {   assert(r > 0.0 && r < 1.0 && p > 0.0 && p <=a);
        setiosflags(ios ::right);
        cout.setf(ios::fixed, ios::floatfiled);
        cout << setprecision(2);
        int m=0; // month number
        r /= 12;  // convert r to a monthly rate
        while(a > 0.0)
        {   cout << setw(8) << m << “.” << setw(12) << a << endl;
            a += r*a; // add interest to remaining balance
            a -= p; // subtract monthly payment
            ++m;
        }
    }
```

4.41

```
    bool is_prime divisor(int d, int n)
    {   assert(d >= 2 && d <= n);
        return is_prime(d) && bool(n%d == 0);
    }
```

4.42

```
    void print_literal_tens(int n)
    {   assert(n >= 2 && n <= 9);
        if(n == 2) cout << "twenty";
        else if(n == 3) cout << "thirty";
        else if(n == 4) cout << "forty";
        else if(n == 5) cout << "fifty";
        else if(n == 6) cout << "sixty";
        else if(n == 7) cout << "seventy";
        else if(n == 8) cout << "eighty";
        else cout << "ninety";
    }
    void print_literal_teens(int n)
    {   assert(n >= 0 && n <= 9);
        if(n == 0) cout << "ten";
        else if(n == 1) cout<< "eleven";
        else if(n == 2) cout<< "twelve";
        else if(n == 3) cout<< "thirteen";
        else if(n == 4) cout<< "fourteen";
        else if(n == 5) cout<< "fifteen";
        else if(n == 6) cout<< "sixteen";
        else if(n == 7) cout<< "seventeen";
        else if(n == 8) cout<< "eighteen";
        else cout << "nineteen";
    }
    void print_dollars(float x)
    {   assert(x >= 0.00 && x < 1000.00);
        int hundreds = int(x)/100;
        int tens = int(x)%100/10;
        int ones = int(x)%10;
        int cents = int (100*x)%100;
        if(hundreds >= 1)
        {   print_literal_digit(hundreds);
            cout << " hundred ";
        }
        if(tens >= 2)
        {   print_literal_tens(tens);
            if(ones >= 1)
            {   cout << "-";
                print_literal_digit(ones);
            }
        }
        else if (tens == 1) print_literal_teens(ones);
        else if (ones >= 1) print_literal digit(ones);
        if(x < 1.0) cout << "no";
```

```
    cout << " dollars and " << cents << " cents”;
}
```