Chapter 3

Control Structures

3.1 BLOCKS AND SCOPE

A *block* is a sequence of statements enclosed in braces. It can be used wherever a single statement can be used. When used in place of a single statement it is also called a *compound statement.*

EXAMPLE 3.1 A Statement Block

Enclosing the statements within braces forms a statement block:

```
{   int n;
    cin >> n;
    cout << 2*n << endl;
}
```

The *scope* of an object is that part of the program where the object may be used. If it is declared within a block, its scope extends from the point where it is declared to the end of the innermost block that contains it. In this case we say it is *local* to the block that defines its scope. If an object's declaration is outside all blocks, then its scope extends from its point of declaration to the end of the file in which it is declared. In this case the object is called a *global* object.

No two objects may have the same name within the same block. However, two objects may have the same name within separate blocks. If one block is nested inside another, any object declared in the inner block masks any object with the same name that is declared in the outer block. So the scope of the object declared outside of a block may have holes in it.

EXAMPLE 3.2 Nested Blocks

Here is a complete `C++` program that contains three blocks, one which serves as the body of the `main ()` function, and the other two nested inside the first.

```
int main()
{  int x = 22;
   {  int x = 44;
      cout << x << endl;    // prints 44
   }
   cout << x << endl;       // prints 22
   {  int x = 66;
      cout << x << endl;    // prints 66

   }
   cout << x << endl;       // prints 22
}
```

The output of this program is

```
44
22
66
22
```

Note that this program contains three different objects, all named `x.` Their scopes are determined by the blocks within which they are defined. The first `x` has scope that extends throughout the `main()` block except where it is overridden by more local objects with the same name. The second `x` is local to the inner block where it is declared, and within that scope it masks the scope of the first `x.` So any reference to `x` within that limited scope will be interpreted as referring to the second `x.` Similarly, the scope of the third `x` within its inner block also masks the scope of the first x.

Recall that, for brevity, we have omitted from the examples the two lines

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

for Standard C++ programs, or the equivalent single line

```
#include <iostream.h>
```

for pre-Standard `C++`programs. These are required for any program that uses `cin` or `cout.`

3.2 NAMESPACES

A *namespace* is a named block that is used to express a logical grouping of statements within a program. ISO Standard C++ allows namespaces. The block name is simply an identifier declared with the `namespace` key word at the head of the block:

`namespace` *`block-name block`*

It is then used outside the block to allow external reference to the names declared inside the block:

`block-name``::`*`name`*

The double colon : : is called the *binary scope resolution operator*.

EXAMPLE 3.3 Using Namespaces to Access Names from Outside Their Scopes

```
namespace Blockl
{   int x = 44;
}
namespace Block2
{   int x = 66;
}
int main()
{
    int x = 22;
    cout << "In main(), x = " << x << endl; // prints 22
    cout << "In namespace Block1, x = " << Block1::x << endl;  // prints 44
    cout << "In main(), x = " << x << endl; // prints 22
    cout << "In namespace Block2, x = " << Block2::x << endl;  // prints 66
}
```

Note the required **`namespace`** keyword preceding each block name.

In addition to allowing out-of-scope access to names defined within namespaces, the scope resolution operator : : can also be used without a block name to refer to global variables.

EXAMPLE 3.4 Using the Scope Resolution Operator to Access Global Names

```
int x = 88;     // x is a global variable
int main()
{   int x = 66;
    cout << x << endl; // prints 66
    cout << ::x << endl; // prints 88
    return 0;
}
```

If there is no conflict in names, scope resolution prefixes can be avoided by means of `using` declarations and directives. Indeed, that is the purpose of the statement

```
using namespace std;
```

that we have been assuming in all of our examples. This simply alerts the compiler to that fact that the names `cin` and `cout` (and others) are defined in the `std` namespace that is declared in the `<iostream>` file. Without that statement, every reference to `cin` would have to be `std::cin,` and every reference to `cout` would have to be `std::cout.`

EXAMPLE 3.5 Resolving Scope with a using Directive

```
namespace X
{   int x = 22;
}
namespace Y
{   int y = 33;
    namespace Z
    {   int z = 44;
    }
}
int main()
{   int x = 55;
    cout << X::x << " " << Y::Z::z << " " << x;
    using namespace Y;               // after this, Y:: can be omitted
    cout << y << "" << Z::z << endl;
}
```

The output is

```
22 44 55 33 44
```

Note that it would be risky to include the directive **`using namespace`** `X;` within `main()` because then any reference to `x` would be ambiguous.

3.3 THE `if` AND `if...else` STATEMENTS

The **`if`** statement is used for conditional execution. Its syntax is

```
if (condition) statement;
```

The `statement` will be executed only if the `condition` is `true`. The parentheses around the `condition` are required.

EXAMPLE 3.6 Using an if Statement

```
int n;
cin >> n;
if (n > 2) cout << "ok. Thanks. ";
cout << "Goodbye.\n";
```

If the user inputs an integer that is greater than 2, then the output will be

```
ok. Thanks. Good-bye.
```

Otherwise, the output will be only

```
Good-bye.
```

Note that only the statement that immediately follows the condition is part of the **`if`** statement. The statement that follows it is independent of the condition and executes regardless of whether `n` > 2. Also note that the condition `n > 2` must be enclosed within parentheses:

```
if n == 4 cout << "ok. Thanks. "; // ERROR: missing parentheses
```

EXAMPLE 3.7 Using a Block in an `if` Statement

```
if (n > 20)
{   cout << "That is too big! Enter a smaller n: ";
    cin >> n;
}
cout << "Thank you\n";
```

This executes both statements between the braces if `n` is greater than 20.

The **`if`**`...`**`else`** statement is the same as the **`if`** statement with an appended **`else`** clause: **`if (`***condition) statementl;* **`else`** *statement2;*

If the `condition` is `true`, only `statementl` is executed; otherwise only `statement2` is. The semicolon that precedes the `else` is required.

EXAMPLE 3.8 Using an `if...else` Statement

```
if(n%2 == 0) cout << "n is even\n";
else cout "n is odd\n";
```

The value of the expression `n%2` is the remainder from dividing `n` by 2; that will be `0` if `n` is even, or `1` if `n` is odd. So this **`if`**`...`**`else`** statement will print `n is even` only if `n` is even, and it will print `n is odd` only if `n` is odd.

Like blocks, **`if`** statements and **`if...else`** statements can be used anywhere that single statements can be used. This allows nested conditionals:

EXAMPLE 3.9 Nested Conditionals

```
if(n > 5)
    if(n > 8) cout << "n > 8" << endl;
    else cout << "5 < n <= 8" << endl;
else
    if(n > 2) cout << "2 < n <= 5" << endl;
    else cout << n <= 2" << endl;
```

This will execute exactly one of the four output statements, depending upon which conditions are true.

It is usually better to avoid nested conditionals like the one in Example 3.9 because the logic can be confusing. An alternative is to use compound conditions:

EXAMPLE 3.10 Using Compound Conditions

```
if(n > 8)        cout << "8 < n"      << endl;
if(n > 5 && n <= 8) cout << "5 < n <= 8" << endl;
if(n > 2 && n <= 5) cout << "2 < n <= 5" << endl;
if(n <= 2)       cout << "n <= 2"     << endl;
```

This makes the logic much easier to follow, and consequently less prone to error.

A third alternative is a series of nested **`if...else`** statements in a specialized form, called **`else if`** forms:

EXAMPLE3.11 Using Sequential `else if` Forms

```
if(n > 8)      cout << "8 < n"      << endl;
else if(n > 5) cout << "5 < n <= 8" << endl;
else if(n > 2} cout << "2 < n <= 5" << endl;
else           cout << "n <= 2"     << endl;
```

This looks very similar to the code in Example 3.10, and the logic is just as easy to follow. But it is actually simpler. Note that, to be consistent with general nested conditionals, it could have been indented like this:

```
if(n > 8) cout << "8 < n" << endl;
else
    if(n > 5) cout << "5 < n <= 8" << endl;
    else
       if(n > 2) cout << "2 < n <= 5" << endl;
       else cout << "n <= 2" << endl;
```

But the first format is more succinct, just as clear, and more widely used.

3.4 THE CONDITIONAL EXPRESSION OPERATOR

Among its many operators, C++ provides one ternary operator, called the *conditional expression operator*. It is simply an abbreviated alternative to a special case of the **`if...else`** statement. The syntax for the conditional expression operator is

`condition ? expressionl : expression2`

The resulting value of this expression is either *`expressionl`* or expressions according to whether the `condition` is `true` or `false`.

EXAMPLE 3.12 Using the Conditional Expression Operator in an Assignment

The statement

```
abs = (x >= 0 ? x : -x);
```

has the same effect as the statement

```
if(x >= 0) abs = x;
else abs = -x;
```

It assigns `x` to `abs` if `x >= 0;` otherwise it assigns `-x` to `abs.`

The conditional expression operator is useful whenever one of two alternative values is needed.

EXAMPLE 3.13 Using the Conditional Expression Operator in an Output Statement

```
cout << "You " << (x >= 60 ? "passed" : "failed") << " the test.\n";
```

If `x > 60`, this prints `You passed the test.` Otherwise it prints `You failed the test.`

3.5 OPERATORS

An *operator* is a built-in function that is called by means of a special symbol that replaces the usual function notation. The *arithmetic operators* are familiar examples:

```
x = z - y;
y = x*z;
z = x/y;
```

Like most operators, these are *binary operators*, combining two *operands.* ISO Standard C++ defines 68 different built-in operators, and the Standard Library defines many more. These are categorized according to how they are used.

The five binary *arithmetic operators* are: `+`, `-`, `*`, `/`, and `%`, pronounced "plus," "minus," "times," "divided by," and "modulo." There are also six *unary* arithmetic operators, that are used like this:

```
y = +x;      // same as y = x;
y = -x;      // same as y = -1*x;
m = ++n;     // same as m = n = n + 1;
m = --n;     // same as m = n = n – 1;
m = n++;     // same as m = n; n = n + 1;
m = n--;     // same as m = n; n = n – 1;
```

Note that the unary plus operator is really useless since it does not change the value of its operand.

The six *assignment operators* =, +=, -=, *=, /=, and %= were described in section 2.5. The six *relational operators* are ==, !=, <, <=, >, and >=. These are used in conditions (boolean expressions) such as

```
if(x >= y) y = x;
```

Conditional statements also use the *logical operators* (also called *boolean operators*) *&&*, ||, and !, pronounced "and", "or", and "not."

The three logical operators are defined by the following *truth tables:*

p	q	p && q
true	true	true
true	false	false
false	true	false
false	false	false

p	q	P || q
true	true	true
true	false	true
false	true	true
false	false	false

p	*!p*
true	false
false	true

EXAMPLE 3.14 Using Logical Operators

```
if(x > 2 && x < 6) cout << "x is between 2 and 6" << endl;
if(x == 2 || x > 6) cout << "x is 2 or greater than 6" << endl;
if(!(x > 6)) cout << "x is not greater than 6" << endl;
```

Each of these tables is read from left to right. For example, the second table shows that if p is true and q is false, then p || q is true.

Each of the two binary logical operators (&& and ||) will evaluate the first operand and then use that value to determine whether to evaluate the second operand. If p is false, then p && q will return false without even evaluating q. Similarly, if p is true, then p || q will return true without even evaluating q. This is called *short-circuiting.* It is quite useful in certain circumstances.

EXAMPLE 3.15 Depending upon Short Circuiting

```
if(d >= 0 && sqrt(d) < y) cout << (-b + sqrt(d))/(2*a);
```

The call sqrt(d) will crash if d < 0. But because of short circuiting, the condition sqrt(d) < y will not even be evaluated unless the condition d >= 0 is true.

In a complex expression involving several operators, such as `m = 2 * n - rain/3 + 5 *sqrt (X : :x*Y: :y);` the order in which the operators are evaluated is affected by their ***order of precedence.*** Among those operators already considered, that order is:

Operator Category	Operators
scope	::
function, post increment, post decrement	(), ++, --
pre increment, pre decrement, not, unary minus	-+, --, !, -
multiply, divide, modulo	*, /, %
add, subtract	+, -
input, output	>>, <<
less than, greater than	<, <=, >, >=
equal, not equal	==, !=
assignment	=, +=, -=, *=, /=, %=
conditional expression	? :

Grouping by parentheses overrides these rules. For example in the expression a*(b + c), the sum is evaluated before the product. Except for the assignment operator, all of these operators are *left associative*; this means that an expression like x/y*z is evaluated from left to right: (x/y)*z. The assignment operators are exceptions to this rule. When they are chained, as in

```
Z += y = x *= 2;
```

the evaluation is done from right to left: z += (y = (x *= 2)); So if the values of z, y, and x are initially 10, 7, and 3, then this statement will change the value of x to 6, then change the value of y to 6, then change the value of z to 16. Check this!

3.6 THE `while` STATEMENT

The `while` statement repeats the execution of a statement while its control condition is true. Its syntax is

`while` (*condition*) `statement;`

The system will repeatedly evaluate the `condition` and execute the `statement` until the `condition` is false. Of course the `statement` may be a block, or any other kind of statement, including another **`while`** statement. Note that if the `statement` is a block, the `condition` is evaluated only at the beginning of each iteration, before the statements in the block are executed. So on each iteration, all the statements within the block are executed regardless of whether the `condition` becomes false during their execution.

EXAMPLE 3.16 Using a `while` Statement to Add the First 100 Squares

This code adds $1*1+2*2+3*3+\cdots+100*100$ and then prints the sum:

```
int n = 1;
int sum = 0;
while(n <= 100)
{   sum += n*n;
    ++n;
}
cout << sum << endl; // prints 338350
```

The loop iterates 100 times, once for each value of `n` from 1 to 100. After accumulating 100*100 into `sum`, it increments `n` to 101. Then the control condition `(n <= 100)` is `false`, which stops the loop. After that, the statement that follows the loop executes, printing the sum 338,350.

The following is equivalent to the above code:

```
int n = 0;
int sum = 0;
while (++n <= 100)
    sum += n*n;
cout << sum << endl; // prints 338350
```

By initializing `n` at 0 instead of 1, we can use the pre-increment operator `++n` inside the control condition instead of in the loop's block.

Note that the control condition of a `while` loop is evaluated only once for each iteration, at the beginning of the iteration before any of the statements in the loop's block are executed.

EXAMPLE 3.17 Controlling a Loop Interactively with the End-of-File Signal

```
float x;
float sum = 0.0;
while(cin >> x)
    sum += x;
cout << "The sum is " << sum << endl;
```

This loop will continue to iterate as long as values are input for x. The loop stops when the system detects the *end-of-file* signal. On a UNIX or Windows system, that signal is transmitted by pressing `<Ctrl-D>`.

Here is a sample run:

```
5.5 8.8 1.1 4.4
<Ctr1-D>
The sum is 19.8
```

(Note that `<Ctrl-D>` is entered by holding down the **Ctrl** key and pressing the **D** key.)

This works because an input expression such as `cin >> x` returns the `bool` value `true` if and only if the input is successful. When the end-of-file signal is received, the input fails and the expression evaluates to `false`, which stops the loop.

Note that since any type of input failure will stop the loop, the same result could be obtained from the input

```
5.5 8.8 1.1 4.4 STOP!
```

Here, the `S` would stop the loop because only numbers are legal input for the numeric variable `x`.

EXAMPLE 3.18 Controlling a Loop Interactively with a Sentinel

```
char c = ' ';
while(c != '\n')
{   cin.get(c);
    if(c >= 'a' && c <= 'z')
        c = char(c - 'a' + 'A');
   cout.put(c);
}
```

This will read individual characters from the keyboard until `<Return>` key is pressed. Within the loop, the character that are lowercase letters are capitalized, and every character is echoed to the output stream `cout`.

After the loop terminates, the contents of the `cout` buffer are sent to the screen.

Here is a sample run:

```
We hold these Truths to be self-evident
WE HOLD THESE TRUTHS TO BE SELF-EVIDENT
```

The first line (in boldface) was the input.

A special input value like `'\n'` that is used to terminate an input loop is called a *sentinel.*

3.7 THE `do...while` STATEMENT

In some cases, it is preferable to execute a loop's statement once before its control condition is evaluated. That can be done with a **`do...while`** loop. Its syntax is

`do` *`statement`* **`while`** (*`condition`*);

It works the same way as a **`while`** loop except that the condition is evaluated after the statement (or statement block) is executed instead of before.

EXAMPLE 3.19 A do Loop for User-Friendly Input

```
bool cont;
char ans;
do
{   cout << "Continue? (y/n): ";
    cin >> ans;
    if(ans == 'y' || ans == 'Y') cont = true;
    else if (ans == 'n' || ans == 'N') cont = false;
    else cout << "Please answer either y or n.\n";
} while(ans != 'y' && ans != 'Y' && ans != 'n' && ans != * N');
```

This loop will repeatedly ask for an answer until one of the four letters `y, Y, n,` or `N` is input.

EXAMPLE 3.20 An Interesting Number Sequence

Here is a complete C++ program that produces number sequences whose lengths are rather unpredictable:

```
#include <iomanip>                  // defines setw() and int
#include <iostream>
using namespace std;
main()
{   int n;
    cout << "Enter a positive integer: ";
    cin >> n;
    int count = 0;
    do
    {   if(n%2 == 0) n /= 2;
        else n = 3*n + 1;
        cout << setw(6) << n;   // use a field of 6 columns per number
        ++count;
        if(count%10 == 0) cout << endl;   // prints 10 numbers per line
    } while(n > 1);
    cout << "\nThat sequence has " << count << " terms.\n";
}
```

Here are some sample runs. (The dollar sign is the UNIX prompt.)

```
$ ex0320
Enter a positive integer: 3
    10     5    16     8     4     2     1
That sequence has 7 terms.
$ ex0320
Enter a positive integer: 13
    40    20    10     5    16     8     4     2     1
That sequence has 9 terms.
$ ex0320
```

```
Enter a positive integer: 23
    70    35   106    53   160    80    40    20    10     5
    16     8     4     2     1
That sequence has 15 terms.
$ ex0320
Enter a positive integer: 31
    94    47   142    71   214   107   322   161   484   242
   121   364   182    91   274   137   412   206   103   310
   155   466   233   700   350   175   526   263   790   395
  1186   593  1780   890   445  1336   668   334   167   502
   251   754   377  1132   566   283   850   425  1276   638
   319   958   479  1438   719  2158  1079  3238  1619  4858
  2429  7288  3644  1822   911  2734  1367  4102  2051  6154
  3077  9232  4616  2308  1154   577  1732   866   433  1300
   650   325   976   488   244   122    61   184    92    46
    23    70    35   106    53   160    80    40    20    10
     5    16     8     4     2     1
That sequence has 106 terms.
```

3.8 THE for STATEMENT

The **for** statement is the third kind of loop provided by C++. Its syntax is

for(*initializer*; *condition*; *expression*) *statement;*

The initializer executes first (and only then). Then the loop repeatedly evaluates the condition and, if it is true, executes the statement and then the expression.

EXAMPLE 3.21 Using a for Loop to Add the First 100 Squares

This code does the same as that in Example 3.16

```
int sum = 0;
for(int n=l; n <= 100; n++)
    sum += n*n;
cout << sum << endl; // prints 338350
```

The **for** loop is more succinct than the **while** loop and the **do**... **while** loop; its three-part control mechanism includes the initialization, the continuation condition, and the update expression that have to be done separately in the other two loop forms.

Note that a variable may be declared within the initializer of a for loop. If so, ISO Standard C++ restricts the scope of the variable to the statement of the loop. The variable cannot be used outside of the loop:

```
for(int x=0; x<20; x++)
   cout << x*x + x + 41 << endl;
cout << x << endl;                    // ILLEGAL: x is out of scope!
```

Note that in many pre-Standard C++ compilers, this restriction is not observed.

3.9 THE break AND continue STATEMENTS

All three standard loop forms, the **while** loop, the **do**...**while** loop, and the **for** loop, evaluate their continuation condition only at the beginning or end of each iteration. If the body of the loop is a block of statements, the loop will execute them all before evaluating its control condition again. This inflexibility can be overridden with the **continue** and the **break** statements.

The **continue** statement terminates the current iteration without executing the rest of the statements in the loop block; control goes back immediately to the control condition to determine whether to terminate the loop itself or continue with the next iteration.

EXAMPLE 3.22 Using the continue Statement

```
int n =7;
```

```
for(int k=50; k<75; k++)
{   if(k%n == 0) continue;
    cout << k << " ";
}
cout << endl;
```

Here is the output from this code:

```
50 51 52 53 54 55 57 58 59 60 61 62 64 65 66 67 68 69 71 72 73 74
```

Note that the multiples of `7 (56, 73, and 7)` are missing from the output. That is because when `k%7 == 0`, the **`continue`** statement skips over the remaining output statement in the body of the loop.

The **`break`** statement has the same effect as the **`continue`** statement inside a loop's block except that it not only skips the remainder of the current iteration but it also immediately terminates the loop itself, going immediately to the next statement that follows the loop.

EXAMPLE 3.23 Using the break Statement

```
int n =7;
for(int k=50; k<75; k++)
{   if(k%n == 0) break;
    cout << k << " ";
}
cout << endl;
```

Here is the output from this code:

```
50 51 52 53 54 55
```

This is the same as Example 3.22 except that the **`break`** statement immediately terminates the loop the first time that the condition `(k%7 == 0)` is true.

In some cases, it is easier to control loops entirely with **`break`** statements. In these cases, the built-in control condition is set to **`true`**, or simply omitted in **`for`** loops.

EXAMPLE 3.24 Controlling an Infinite `while` Loop

```
int n =4;
while(true)
{   cout << 1.0/n << '\t';
    if(n == 9) break;
    ++n;
}
cout << '\n' << n << endl;
```

The output is:

```
0.25    0.2    0.166667          0.142857          0.125        0.111111
9
```

On the sixth iteration, when `n` is 9, the loop prints `0.111111` and then terminates before incrementing `n` again. The control condition is `true`, so without the break statement the loop would iterate forever.

EXAMPLE 3.25 Controlling a "forever" Loop

```
int product=l, factor, count=0;
cout << "Enter factors. Terminate with 0: ";
for(;;)
{   cin >> factor;
    if(factor == 0) break;
    product *= factor;
   ++count;
}
cout << "The product of the " << count << " factors is " << product << endl;
```

Here is a sample run:

```
Enter factors. Terminate with 0: 3 5 7 9 0
The product of the 4 factors is 945
```

The `for(;;)` construct sets up an infinite loop, just like the **`while`**`(`**`true`**`)` form. In this example, the input value 0 is used as a sentinel. But since the inputs are used as multipliers, the 0 value should not be used after it is input. The `break` statement prevents the sentinel from being factored in or counted. Bjarne Stroustrup suggests that the form `for(;;)` be pronounced "forever."

3.10 LOOP INVARIANTS

Loops are an essential part of any software system. They are also a major source of logical errors. Loop invariants are an effective strategy against these errors.

A *loop invariant* is an assertion about the variables in a loop that is intended to be true at the beginning of each iteration of the loop and after the loop has terminated.

EXAMPLE 3.26 Using a Loop Invariant

Here is some code that computes the power $y = x^n$:

```
double y = 1.0;
for(int i=0; i<n; i++)
    // INVARIANT: y == x*x*...*x (i times)
    y *= x;
```

The loop invariant is expressed as a comment at the beginning of the loop. It asserts that $y = x \cdot x \cdots x$ (i times). If that assertion is true at the end of the loop, then the code is correct: it does compute $y = x^n$. So to verify that the code really is correct, we need only check it against the loop invariant.

When the first iteration begins, $y = 1$ and $i = 0$, so the invariant $y = x \cdot x \cdots x$ (i times) is true. Next, if we assume that the invariant was true at the beginning of some (unspecified) iteration then we can deduce that it is also true at the beginning of the next iteration from the fact that during that iteration y was multiplied by x one more time. It follows then, by the *Principle of Mathematical Induction*, that the loop invariant is indeed true at the beginning of every iteration. This proves conclusively that the code is correct.

EXAMPLE 3.27 Multiplying in *O*(lg *n*) Time

The power function in Example 3.26 is implemented with the standard algorithm: to raise *x* to the power *n*, simply multiply *x* by itself *n* times. The following is a more efficient (but less obvious) method for computing the power function:

```
double y = 1.0, z = x;
int i = n;
while(i > 0)
    // INVARIANT: y*pow(z,i) == pow(x,n)
    if(i%2 == 0) // i is even
    {   z *= z;  // square z
        i /= 2;  // halve
    }
    else // i is odd
    {   y *= z;
        --i;
    }
```

Here is a trace of the execution of this code in computing $y = 3^{10}$:

z	y	i	z^i	$y \cdot z^i$
3	1	10	59,049	59,049
9	1	5	59,049	59,049
9	9	4	6,561	59,049
81	9	2	6,561	59,049
6561	9	1	6.561	59,049
6561	59,049	0	1	59,049

When the loop ends, $y = 59{,}049 = 3^{10}$.

The loop invariant is $y \cdot z^i = x^n$. This is discovered from the trace: $y \cdot z^i$ is always 59,049. We can use this to prove that the code is correct. If the invariant is true after the loop terminates, then $y = y \cdot 1 = y \cdot z^0 = y \cdot z^i = x^n$, since $i = 0$ at that time. When the loop begins, the invariant is true because $y = 1$, $z = x$, and $i = n$. To show that the invariant is true at the beginning of every iteration of the loop, we use mathematical induction again. Suppose that $y \cdot z^i = x^n$ at the beginning of some iteration. If i is even, then z will be squared and i will be halved on that iteration, so the equation $y \cdot z^i = x^n$ will still balance when that iteration has finished (because the value of $y \cdot z^i$ remains *invariant*). Similarly, if i is odd, then y will be multiplied once by x and i will be decremented once, so again $y \cdot z^i$ is invariant and the equation balances at the end of the iteration.

The symbol $O(\lg n)$ ("order lg *n*") is used to describe this implementation of the power function because it has the property that its running time is proportional to the logarithm of the power *n.* Here, lg is the *binary logarithm* (base 2), which is the logarithm usually used in computer science. It is proportional to other logarithms.

3.11 NESTED LOOPS

A **for** statement (or a **while** statement or a **do...while** statement) can be used anywhere that a simple statement can be used, including inside another loop. Such combinations are called *nested loops.*

EXAMPLE 3.28 Printing a Multiplication Table

This program uses two nested for loops to print a multiplication table:

```
#include <iomanip>
#include <iostream>
using namespace std;
int main()
{   setiosflags(ios::right);
    int n;
    cout << "How many columns? (1-16): ";
    cin >> n;
    for(int x=l; x <= n; x++)
    {   for(int y=l; y <= n; y++) cout << setw(5) << x*y; cout << endl;
    }
}
```

Here is a sample run:

```
How many columns? (1-16): 12
    1    2    3    4    5    6    7    8    9   10   11   12
    2    4    6    8   10   12   14   16   18   20   22   24
    3    6    9   12   15   18   21   24   27   30   33   36
    4    8   12   16   20   24   28   32   36   40   44   48
    5   10   15   20   25   30   35   40   45   50   55   60
    6   12   18   24   30   36   42   48   54   60   66   72
    7   14   21   28   35   42   49   56   63   70   77   84
    8   16   24   32   40   48   56   64   72   80   88   96
    9   18   27   36   45   54   63   72   81   90   99  108
   10   20   30   40   50   60   70   80   90  100  110  120
   11   22   33   44   55   66   77   88   99  110  121  132
   12   24   36   48   60   72   84   96  108  120  132  144
```

Note the essential block inside the outer loop. It contains two independent statements: the inner loop and the last `cout` statement. Each iteration of the outer loop prints one line.

When a **break** or **continue** statement is used in a nested loop, it interrupts only its closest containing loop; the outer loop(s) are unaffected.

EXAMPLE 3.29 Using a `break` Statement in a Nested *Loop*

```
for(int i=l; i<5; i++)
    for(int j=l; j<5; j++)
        for(int k=l; k<5; k++)
            if(i + j + k > 5) break;
            else cout << '\t' << I << '\t' << j << '\t' << k << endl;
```

The out is

```
1       1       1
1       1       2
1       1       3
1       2       1
1       2       2
1       3       1
2       1       1
2       1       2
2       2       1
3       1       1
```

Review Questions

3.1 What is a compound statement?

3.2 What is scope of a variable?

3.3 What is a namespace?

3.4 What does a `using` directive do?

Problems

3.5 List by line number the scope of each variable declared in the following program. Then give its output:

```
namespace loc1                                                          // 1
{   int x = 44;                                                         // 2
    int z = 55;                                                         // 3
}                                                                       // 4
namespace loc2                                                          // 5
{   int x = 77;                                                         // 6
    int z = 88;                                                         // 7
}                                                                       // 8
int x = 22;                                                             // 9
int main()                                                              // 10
{   int y = 33;                                                         // 11
    cout << x << " " << y << endl;                                      // 12
    cout << loc1::x << " " << loc1::y << " " << loc1::z << endl;        // 13
    int x = 66;                                                         // 14
    cout << loc2::x << " " << loc2::y << " " << loc2::z << endl;        // 15
    cout << x << " " << y << endl;                                      // 16
}                                                                       // 17
```

3.6 In the following program, replace the comments with statements that perform the task:

```
int n = 33;
namespace blockl
{   int n = 77;
}
namespace block2
{   int n = 99;
}
int main()
{   int n = 55;
    // print the n whose value is 33
    // print the n whose value is 55
    // print the n whose value is 77
    // print the n whose value is 99
}
```

3.7 Find the error in each of the following code fragments:

a.
```
if n < 0 cout << "n is negative\n";
```

b.
```
if(n < 0) cout << "n is negative\n"
else cout << "n is non-negative\n" ;
```

c.
```
if(n < 0) cout << "n is negative\n";
    cout << "Is that ok?\n";
else cout << "n is non-negative\n";
```

d.
```
if(n < 0) cout << "n < 0";
if(n == 0) cout << "n == 0";
else cout << "n > 0";
```

3.8 How many numbers will be printed in each of the following if the value of n is 22?

a.
```
if(n < 20) cout << n << " ";
cout << 2*11 << endl;
```

b.
```
if(n > 20) cout << n << " ";
cout << 2*n << endl;
```

c.
```
if(n < 20) cout << n << " ";
else cout << 2*n << " ";
cout << 3*n << endl;
```
d.
```
if(n > 20)
{   cout << n << " ";
    cout << 2*n << " ";
}
else
{   cout << 3*n << " ";
    cout << 4*n << " ";
}
cout << -n << endl;
```

3.9 Find the error in each of the following loops:

a.
```
while(n < 20);
    cout << n++ << endl;
```
b.
```
for(int i = l, i <= 8, i++)
    cout << 1.0/i << endl;
```
c.
```
int n = 10;
do
    cout << 1.0/n;
    ++n;
while (n < 20);
```

d.
```
for(int i = 10; i<20; i--)
    cout << i*i << endl;
```

Programming Problems

3.10 A year is a leap year if it is divisible by 4 but not by 100 unless it is also divisible by 400. So the years 1996 and 2000 are leap years, but the years 1999 and 1900 are not. Write a program that inputs a year and prints whether it is a leap year.

3.11 Implement the quadratic formula to solve the quadratic equation $ax^2 + bx + c = 0$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Be sure to handle the special cases: where $a = 0$, where the two roots are equal, and where the discriminant $b^2 - 4ac$ is negative.

3.12 Write a program that inputs a day number of the year and prints the month and day of the month. Assume that the year is not a leap year.

3.13 Use the conditional expression operator to translate the following code into a single assignment statement:
```
if (x >= 0) sqrtx = sqrt(x);
else sqrtx = sqrt(–x);
```

3.14 Use **if...else** statements to remove the conditional expression operators from
```
sgn = ( x >= 0 ? ( x > 0 ? 1 : 0 ) : -1 );
```

3.15 The (integral) *binary logarithm* of a positive number is the number of times it can be divided in two until the result is less than 2. That is, lg *x* is the largest power of 2 that is ≤ x. For example, lg 4 = 2, lg 7.9 = 2, and lg 8 = 3. Write a program that inputs a number and then uses a **while** loop to find and print its binary logarithm.

3.16 Rewrite the following **while** loop as a **for** loop:
```
int i = 4;
while(i < 20)
{   cout << 1.0/i << endl;
    ++i;
}
```

3.17 Rewrite the following `for` loop as a **while** loop:
```
for(int j  =22;  j > ;  j--)
    sum += log(j*j);
```

3.18 The program in Problem 2.24 on page 38 is prone to integer overflow. For example, if $n = 9$, the initialization

```
int round = int(x + 0.5);
```

fails because `x + 0.5 = 3,141,592,634.089793,` so that `int(x + 0.5)` = 3,141,592,634. The largest value for 32-bit integers is 2,147,483,647. Re-write this program so that it works for values $0 \le n \le 15$.

3.19 The sum of the first *n* positive integers $(1 + 2 + 3 + \cdots + n)$ is given by the formula:

$$\sum_{i=1}^{n} i = \frac{n(n-1)}{2}$$

Write a complete program that checks this formula by inputting *n* and then computing and comparing the values of both sides of this equation.

3.20 The sum of the first *n* squares $(1 + 4 + 9 + \cdots + n^2)$ is given by the formula

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Write a complete program that checks this formula by inputting *n* and then computing and comparing the values of both sides of this equation.

3.21 The sum of the first *n* cubes $(1 + 8 + 27 + \cdots + n^3)$ is given by the formula

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n-1)^2}{4}$$

Write a complete program that checks this formula by inputting *n* and then computing and comparing the values of both sides of this equation.

3.22 The *factorial function* is defined by the formula $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$; that is, "*n* factorial" is the product of the first *n* positive integers. For example, 5! = 120 because $120 = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$. Write a complete program that inputs *n* and then computes and prints the value of *n*!.

3.23 The *permutation function* is defined by the formula $p(n,k) = n(n-1)(n-2)\cdots(n-k+1)$; that is, $p(n,k)$ is the product of the *k* largest integers that are $\le n$. For example, $p(11,4) = 11 \cdot 10 \cdot 9 \cdot 8 = 7920$. Write a complete program that inputs *n* and *k* and then computes and prints the value of $p(11,4)$.

3.24 The *combination function* is defined by the formula $c(n,k) = \left(\frac{n}{1}\right)\left(\frac{n-1}{2}\right)\cdots\left(\frac{n-k+1}{k}\right)$; that is, *c*(*n*,*k*) is the product of the *k* ratios *j*/*i*, where *j* ranges from *n* down to *n* – *k* + 1 and *i* ranges from 1 up to *k*. For example, $c(11,4) = \left(\frac{11}{1}\right)\left(\frac{10}{2}\right)\left(\frac{9}{3}\right)\left(\frac{8}{4}\right) = 11 \cdot 5 \cdot 3 \cdot 2 = 330$. Write a complete program that inputs *n* and *k* and then computes and prints the value of $(11,4)$. (Note that *c*(*n*,*k*) is always a positive integer.)

3.25 Write a complete program that inputs a positive integer *n* and then prints a triangle of asterisks *n* lines high and 2*n* – 1 columns wide. For example, if the input is 5 then the output should be

```
    *
   * * *
  * * * * *
 * * * * * * *
* * * * * * * * *
```

3.26 Write a complete program that inputs a positive integer *n* and then prints a rectangle of asterisks *n* lines high and 2*n* columns wide. For example, if the input is 5 then the output should be

```
* * * * * * * * * * *
*                   *
*                   *
*                   *
* * * * * * * * * * *
```

3.27 Write a complete program that implements the following algorithm that computes the greatest common divisor of two given positive integers. For example, if 441 and 252 are input, then the program should print that 63 is their greatest common divisor:

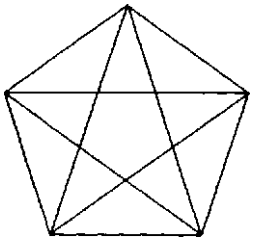
Algorithm 3.1 The Euclidean Algorithm

To find the greatest common divisor (gcd of two positive integers *m* and *n*:

1. Subtract *m* from *n* repeatedly until $n < m$.
2. Interchange the values of *m* and *n*.
3. Repeat steps 1-2 until $m = 0$.
4. Then *n* is the greatest common divisor of the two original numbers.

This is Proposition 2 in Book VII of Euclid's *Elements*, written around 300 B.C.

3.28 The *Fibonacci numbers* are those in the sequence 0, 1, 1, 2, 3, 5, 8,... in which each successive number is obtained by adding its two predecessorrite a complete program that inputs a positive integer *n* and then prints the *n* + 1 Fibonacci numbers $f_0, f_1, f_2, \ldots, f_n$. For example, if 6 is input, then it would print 0, 1, 1, 2, 3, 5, 8.



3.29 The *golden mean* is defined to be the constant $\phi = (1 + \sqrt{5})/2 = 1.61803\ldots$. It is the solution to the ancient Greek mathematical question about where to divide a segment so that the ratio of the larger piece to the smaller piece is the same as the ratio of the original segment to the larger piece. The constant plays an important role in geometry, computer science, and art history. For example, each intersection of each line in the mystic pentagram is at the golden mean. Some important Renaissance artists used the ratio of the golden mean to achieve aesthetic balance in their paintings. The ratios of adjacent Fibonacci numbers converge to the golden mean. For example, $f_{10}/f_9 = 55/34 = 1.61765$, and $f_{14}/f_{13} = 377/233 = 1.61803$. Modify your solution to Problem 3.28 so that it also prints the ratios f_k/f_{k-1} for *k* = 2, 3,..., *n*, and then print the decimal value of ϕ for comparison.

3.30 The *least squares method* discovered by Carl Friedrich Gauss (1777-1855) for interpolating data produces the following formulas for the *regression line* that best fits a given set of data points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$

$$s_x = \sum_{i=1}^{n} x_i$$

$$s_y = \sum_{i=1}^{n} y_i$$

$$s_{xx} = \sum_{i=1}^{n} x_i^2$$

$$s_{xy} = \sum_{i=1}^{n} x_i y_i$$

$$\bar{x} = \frac{s_x}{n}$$

$$\bar{y} = \frac{s_y}{n}$$

$$m = \frac{n \cdot s_{xy} - s_x \cdot s_y}{n \cdot s_{xx} - s_x \cdot s_y}$$

$$b = \bar{y} - m \cdot \bar{x}$$

Then the equation for the best fit regression line is $y = mx + b$. Write a complete program that inputs the number *n* of data points and then the *n* pairs of numbers x_i, y_i. It should then implement these formulas to obtain and print the regression line equation. Finally, it should ask the user to input other *x* values for which it will then use the equation to compute the corresponding predicted (interpolated) *y* values.

3.31 A loan is *amortized* by printing its schedule of periodic payments along with the remaining balance after each payment. Write a complete program that prints an amortization schedule for a given loan amount a at a given interest rate *r* and a given monthly payment *p*. Your program should input *a*, *n*, and *p*, and then print a series of lines, one for each monthly payment, showing the payment number and the remaining balance after that payment.

Solutions

3.1 A compound statement, also called a block, is a sequence of statements delimited by braces.

3.2 The scope of a variable is that part of the program where it may be used.

3.3 A namespace is a named block.

3.4 A `using` directive simply obviates the need for the scope resolution prefix in references to names outside of their scopes.

3.5 The scope of the x that is defined on line 1 is lines 1–5.
The scope of the `y` that is defined on line 3 is lines 3–11 and 16–17.
The scope of the x that is defined on line 6 is lines 6–9.
The scope of the `z` that is defined on line 7 is lines 7–9.
The scope of the x that is defined on line 10 is lines 10–17.
The scope of the `y` that is defined on line 12 is lines 12–15.
The scope of the `z` that is defined on line 13 is lines 13–15.
The output is

```
22 33
44 33 55
66 77 88
66 33
```

3.6

```
cout << ::n << endl;          // prints the n whose value is 33
cout << n << endl;            // prints the n whose value is 55
cout << block1::n << endl;    // prints the n whose value is 77
cout << block2::n << endl;    // prints the n whose value is 99
```

3.7 ***a.*** Syntax error: missing parentheses around the condition.
b. Syntax error: missing semicolon before the `else`.
c. Compile-time error: an independent statement cannot precede an `else`.
d. Logical error: the `else` is independent of the first condition, so the code will print both `n > 0` and `n < 0` if `n` is negative. Another `else` is needed before the second `if`:

```
if(n < 0) cout << "n < 0";
else if(n == 0) cout << "n == 0";
else cout << "n > 0";
```

3.8 ***a.*** One: `44`
b. Two: `22 4 4`
c. Two: `44 66`
d. Three: `22 44 -22`

3.9 ***a.*** Logical error: the semicolon after the left parenthesis means that the loop statement is empty. That results in an *infinite loop* because the control variable `n` does not change.
b. Syntax error: the three parts of the `for` loop control must be separated by semicolons, not commas.
c. Compile-time error: only one statement or block is allowed between the `do` and the `while`.
d. Logical error: this is an infinite loop because `i` is decreasing and the control condition is `i<20`.

3.10

```
int main()
{   int n;
    cout << "Enter year: ";
    cin >> n;
    if(n % 400 == 0) cout << n << " is a leap year.\n";
    else if(n % 100 == 0) cout << n << " is a not leap year.\n";
    else if(n % 4 == 0) cout << n << " is a leap year.\n";
    else cout << n << " is a not leap year.\n";
}
```

3.11

```
int main()
{   double a, b, c;
    cout << "Enter the coefficients a, b, and c: ";
    cin >> a >> b >> c;
    if(a == 0)
        if(b == 0)
            if(c == 0) cout << "Every real number is a solution";
            else cout << "There are no solutions";
        else cout << "The unique solution is " << -c/b;
    else
    {   double d = b*b - 4*a*c;
        if(d < 0) cout << "There are no real solutions";
        else if (d == 0) cout << "The unique solution is "<<-b/(2*a);
        else
        {   double sqrtd = sqrt(d);
            double xl = (-b + sqrtd)/(2*a);
            double x2 = (-b - sqrtd)/(2*a);
            cout << "The two solutions are " << xl << " and " << x2;
        }
    }
    cout << endl;
}
```

3.12

```
int main ()
{   int day;
    cout << "Enter the day of the year (1-365): ";
    cin >> day;
    if(day < 32) cout << "January ";
    else
    {   day -= 31;
        if(day < 29) cout << "February ";
        else
        {   day -= 28;
            if(day < 32) cout << "March ";
            else
            {   day -= 31;
                if(day < 31) cout << "April ";
                else
                {   day -= 30;
                    if(day < 32) cout << "May ";
                    else
                    {   day -= 31;
                        if(day < 31) cout << "June ";
                        else
                        {   day -= 30;
                            if(day < 32) cout << "July ";
                            else
                            {   day -= 31;
                                if(day < 32) cout << "August ";
                                else
                                {   day -= 31;
                                    if(day < 31) cout << "September";
                                    else
                                    {   day -= 30;
                                        if(day < 32) cout << "October ";
                                        else
                                        {   day -= 31;
                                            if(day < 31) cout << "November ";
                                            else
                                            {   day -= 30;
                                                cout << "December ";
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    cout << day << endl;
}
```

3.13

```
sqrt = ( x >= 0 ? sqrt(x) : sqrt(-x) );
```

3.14

```
if(x > 0) sgn = 1;
else if(x == 0) sgn = 0;
else sgn = -1;
```

3.15

```
int main()
{   float x;
    int n=2;
    int lgx=0;
    cin >> x;
    while(n <= x)
    {   n *= 2;
        ++lgx;
    }
    cout << "lg(" << << x << ") = " << lgx << endl;
```

3.16
```
for(int i = 4; i < 20; i++)
    cout << 1.0/i << endl;
```
3.17
```
int j = 22;
while(j > 8)
{   sum += log(j*j);
    --j
}
```
3.18
```
#include <iomanip>
#include <iostream>
#include <cmath>
using namespace std;
int main()
{   double x = 3.141592653589793;
    cout << setprecision{16) << x << endl;
    int n;
    cout << "Enter number of digits: ";
    cin >> n;
    if(n < 9)
    {   x *= pow(10,n);
        int round = int(x + 0.5);
        x = double(round)/pow(10,n);
    }
    else                                    // e.g., n == 13
    {   x *= pow(10,8);                     // x == 314,159,265.3589793
        int m = int(x);                     // m == 314,159,265
        double y = (x - m)*pow(10,n-8);     // y == 35897.93
        int round = int(y + 0.5);           // round == 35898
        y = double(m)*pow(10,n-8);          // y == 31,425,926,535,898.0
        x = y/pow(10,n);                    // x == 3.1415926535898
    }
    cout << x << endl;
}
```
3.19
```
int main()
{   int n, sum=0;
    cout << "Enter n: ";
    cin >> n;
    for(int i=1; i <= n; i++)
        sum += i;
    cout << "1 + 2 + 3 + ... + n = " << sum << endl;
    cout << "n*(n+l)/2           = " << n*(n+1)/2 << endl;
}
```
3.20
```
main()
{   int n, sum=0;
    cout << "Enter n: ";
    cin >> n;
    for(int i=l; i <= n; i++)
        sum += i*i;
    cout << "1 + 4 + 9 + ... + n*n = " << sum << endl;
    cout << "n*(n+1)*(2*n+l)/6     = " << n*(n+1)*(2*n+1)/6 << endl;
}
```
3.21
```
int main()
{   int n, sum=0;
    cout << "Enter n: ";
    cin >> n;
    for(int i=l; i <= n; i++)
        sum += i*i*i;
    cout << "1 + 8 + 27 + ... + n*n*n = " << sum « endl;
    cout << "n*n*(n+1)*(n+1)/4        = " << n*n*(n+1)*(n+1)/4 << endl;
}
```
3.22
```
int main()
{   int n, fact=l;
    cout << "Enter n: ";
    cin >> n;
    for(int i=2; i <= n; i++)
        fact * = i;
    cout << n << "! = 1*2*...*" << n << " = " << fact << endl;
}
```
3.23
```
int main()
{   int n, k, perm=l;
    cout << "Enter n and k: ";
    cin >> n >> k;
```

```
    for(int i=l; i <= k; i++, n--)
        perm *= n;
    cout << "p(" << n+k << "," << k << ") = " << perm << endl;
}
```

3.24

```
int main()
{   int n, k, comb=l;
    cout << "Enter n and k: ";
    cin >> n >> k;
    for(int i=l; i <= k; i++, n--)
        comb = comb*n/i;
    cout << "c(" << n+k << "," << k << ") = " << comb << endl;
}
```

3.25

```
int main()
{   int n;
    cout << "How many lines? (1-40): ";
    cin >> n;
    for(int i=0; i<n; i++)    // i = number of lines printed
    {   for(int j=l; j <= n + i; j++) // j = current column number
        cout << (j < n - i  ? ' ' : '*' ) ;  // print n - i - 1 blanks
        cout << endl;
    }
}
```

3.26

```
int main()
{   int n;
    cout << "How many lines? (1-40): ";
    cin >> n;
    for(int i=0; i<n; i++)    // i = number of lines printed
    {   for(int j=l; j <= 2*n; j++)  // j = current column number
            cout << (i > 0 && i < n-1 && j > 1 && j < 2*n ? ' * : '**);
        cout << endl;
```

3.27

```
int main()
{   int m, n, tmp, gcd;
    cout << "Enter two positive integers: ";
    cin >> m >> n;
    cout << "The greatest common divisor of " << m << " and " << n;
    do
    {   while(m <= n)
            n -= m;
        tmp = m;
        m = n;
        n = tmp;
    } while(m > 0);
    cout << " is " << n << endl;
}
```

3.28

```
int main()
{   int f0=0, fl=l, f2, n;
    cout << "How many Fibonacci numbers do you want? ";
    cin >> n;
    cout << "\tfl = 1\n";
    for(int i=2; i <= n; i++)
    {   f2 = fl + f0;
        cout << "\tf" << i << " = " << f2 << endl;
        f0 = fl;
        fl = f2;
    }
}
```

3.29

```
int main()
{   setiosflags(ios::right);
    cout.setf(ios::fixed, ios::floatfield);
    cout << setprecision(14);
    int f0=0, fl=l, f2, n;
    cout << "How many Fibonacci numbers do you want? ";
    cin >> n;
    cout << setw(8) << "fl =" << setw(12) << "l\n";
    for(int i=2; i <= n; i++)
    {   f2 = fl + f0;
        cout << setw(5) << "f" << i << " = "
             << (i < 10 ? setw(11) : setw(10)) << f2
             << setw(9) << "f" << i << " / f "  << i-1
                        << (i == 10 ? " " : " " ) << " = "
```

```
                << (i<10? setw(18): setw(16)) « double(f2)/fl << endl;
            fO = fl;
            fl = f2;
        }
        double phi = (1 + sqrt(5)/2;
        cout << " The Golden Mean = (1 +sqrt(5))/2 = " << phi << endl;
    }
```

3.30

```
    int main()
    {   double x, y, sumx=0.0, sumy=0.0, sumxx=0.0, sumxy=0.0;
        int n;
        cout << "Enter the number of data points: ";
        cin >> n;
        cout << "Enter " << n << " pairs x and y:\n";
        set ios flags (ios :: right);
        for(int i=l; i <= n; i++)
        {   cout << setw(8) << i << ": ";
            cin >> x >> y;
            sumx += x;
            sumy += y;
            sumxx += x*x;
            sumxy += x*y;
        }
        double meanx = sumx/n;
        double meany = sumy/n;
        double m = (n*sumxy - sumx*sumy)/(n*sumxx - sumx*sumx);
        double b = meany - m*meanx;
        cout << "The equation of the Gaussian regression line is: y = "
             << m << "x + " << b << endl;
        char ans;
        cout << "Do you want to interpolate? (y/n): ";
        cin >> ans;
        if(ans == 'y' || ans == 'Y')
        {   cout << "Enter x values, one per line.\n"
                 << "Terminate input with <Ctrl-D>:\n";
            while(cin >> x)
                cout << "\ty = " << m*x + b << endl;
        }
    }
```

3.31

```
    int main()
    {   float a; // original amount of loan
        float r; // interest rate; e.g., r = 0.06 for 6%
        float p; // monthly payment
        int m=0; // month number
        float x; // remaining balance after m months
        cout << "Enter amount of loan (e.g., 10000.00): ";
        cin >> a;
        cout << "Enter annual interest rate (e.g., 0.06): ";
        cin >> r;
        r /= 12; // convert r to a monthly rate
        cout << "Enter monthly payment (e.g., 350.00): ";
        cin >> p;
        x = a;
        while(x > 0.0)
        {   cout << m << ",\t" << x << endl;
            x += r*x; // add interest to remaining balance
            x -= p; // subtract monthly payment
            ++m;
        }
    }
```