Iteration with Programs and Classes 5

"What IS the use of repeating all that stuff,"
the Mock Turtle interrupted, "if you don't explain it as you go on?
It's by far the most confusing thing I ever heard!"
Lewis Carroll
Alice's Adventures in Wonderland

I shall never believe that God plays dice with the world.
Albert Einstein
Einstein, His Life and Times, Philipp Frank

The `if/else` statement selects different code fragments depending on values calculated at run time by the program. In this chapter we will study control statements called **loops,** which are used to execute code segments repeatedly. Repetition significantly extends the kinds of programs we can write. We will also study several classes that extend the domain of problems we can solve by writing programs.

To extend the range of problems and programs, we will use some basic design guidelines that help in writing code, functions, and programs. As programs get larger and more complicated, these design guidelines will help in managing the complexity that comes with harder and larger problems.

In the first part of the chapter we'll introduce a basic loop statement. We'll use loops to study applications in different areas of computer science. We'll end the chapter with a study of two classes used in this book that extend the kind of programs you can write. Using loops and these classes will make it possible to write programs to print calendars for any year, to simulate gambling games, and to solve complex mathematical equations.

5.1 The `while` Loop

banana problem: Not knowing where or when to bring a
production to a close. "I know how to spell 'banana,'
but I don't know when to stop."
The New Hacker's Dictionary

In the last chapter Program 4.10, *numtoeng.cpp*, printed English text for integers in the range of 1–99. Converting this program to handle all C++ integer values would be difficult without using loops. Loops are used to execute a group of statements repeatedly. Repeated execution is often called **iteration**. The most basic statement in C++ for looping is the **while** statement. It is similar syntactically to the `if` statement, but very different semantically. Both statements have tests whose truth determines whether a block of

```
if (test)
{
   statement list;
}
next statement;
```

```
while (test)
{
    statement list;
}
next statement;
```

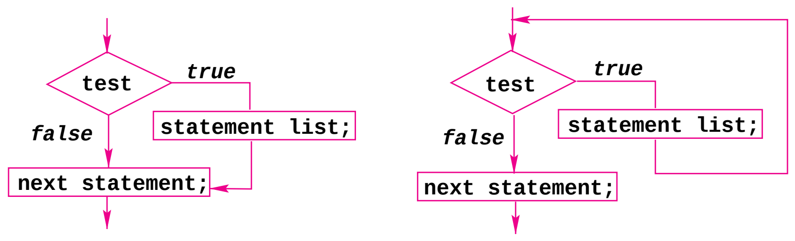


Figure 5.1 Flow control for `if` and `while` statements.

statements is executed. When the test of an `if` statement is true, the block of statements that the test controls is executed once. In contrast, the block of statements controlled by the test of a `while` loop is executed repeatedly, as long as the test is true.

The control flow for `if` statements and `while` statements is shown in Fig. 5.1. In a `while` loop, after execution of the last statement in the **loop body** (the block of statements guarded by the test), the test expression is evaluated again. If it is true, the statements in the loop body are executed again, and the process is repeated until the test becomes false. The test of a loop must be false when the loop exits. The body of a `while` loop is the group of statements in the curly braces guarded by the parenthesized test. The test is evaluated once before all the statements in the loop body are executed, *not* once after each statement. If the test is true, *all* the statements in the body are executed. After the last statement in the body is executed, the test is evaluated again. If the test evaluates to true, the statements in the loop body are executed again, and this process of test/execute repeats until the test is false. (We will learn methods for "breaking" out of loops later that invalidate this rule, but it is a good rule to keep in mind when designing loops.)

Syntax: while statement

`while` (*test expression*)
{
   statement list;
}

When writing loops, remember that the loop test is *not* reevaluated after each statement in the loop body, only after the last statement. To ensure that loops do not execute forever, it's important that at least one statement in the loop changes the values that are part of the test expression. As a simple example, Program 5.1 prints a string backwards.

Program 5.1 revstring.cpp

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

int main()
{
    int k;
    string s;
    cout << "enter string: ";
    cin >> s;
    cout << s << " reversed is ";

    k = s.length() – 1;   // index of last character in s
    while (k >= 0)
    {   cout << s.substr(k,1);
        k –= 1;
    }
    return 0;
}
```

revstring.cpp

OUTPUT

```
prompt> revstring
enter string: desserts
desserts reversed is stressed
prompt> revstring
enter string: deliver
deliver reversed is reviled
```

In Program 5.1 the value of the indexing variable k changes each time the loop executes. Since k is used in the loop guard, and k decreases each time the loop executes, you can reason informally that the loop will terminate: the loop executes exactly as many times as there are characters in the string s. Developing loop tests/guards can be difficult, and we'll study techniques that will help you develop loops that execute correctly. In general there are three conceptual parts in developing a loop test.

1. The **initialization** of variables/expressions that are part of the loop, in particular of the loop guard. In *revstring.cpp* the initialization is the following statement.

```
k = s.length() - 1;   // index of last character in s
```

2. The loop guard or test which is a boolean expression whose truth determines if the loop body executes. This is `k >= 0` in *revstring.cpp*.
3. The **update** of variables/expressions. The update must have the potential to make the loop test false. Usually this means changing the value of a variable used in the test. In *revstring.cpp* the following statement is the update.

```
k -= 1;
```

For the string `"flow"`, the initial value of `k` is 3. The loop body executes for `k` having the values 3, 2, 1, and 0. When `k` is zero, the letter ’f’ is printed, and `k` is decremented to have the value −1. The loop guard is tested and is false, so the loop exits when `k` has the value −1.

5.1.1 Infinite Loops

You must be careful when writing loops, because it is possible for a loop to execute forever—a so-called **infinite loop.** As a simple example, consider the loop

```
while (6 != 4)
{   cout << "this will be printed many times" << endl;
}
```

which will execute forever (or until the user stops the program), because 6 is not equal to 4, and the truth of the loop test is unchanged by any of the statements in the loop body. On many systems, typing Ctrl-`C` will stop an infinite loop.

Usually infinite loops aren’t as easy to spot as the loop above. You may, for example, forget to update a variable and thus create an infinite loop. For example, leaving out the statement `k -= 1` in *revstring.cpp*, Program 5.1 creates an infinite loop that prints the last character of the string “forever”. The following loop is infinite for some values of `num`.

```
int num;
cin >> num;
int start = 0;
while (start != num)
{   start += 2;
}
```

The values for `start` are $\{0, 2, 4, \ldots\}$. If `num` is an odd number, the loop is infinite. If the purpose of the loop is to increment `start` until it “passes” `num`, then it would be better to use the following loop test.

```
int num;
cin >> num;
int start = 0;
while (start <= num)
{   start += 2;
}
```

Pause to Reflect

5.1 Write a loop to print the numbers from 1 up to a value entered by the user, one number per line. Modify the loop to print the numbers from the user-entered value down to 1.

5.2 Complete the following loop so that it prints all powers of two less than 30,000, starting with `1 2 4 8 16` ...You can do this by adding a single `*=` statement to the loop.

```
num = 1;
while (num < 30000)
{   cout << num << endl;

}
```

5.3 How can you determine quickly that the following loop is an infinite loop (and will execute "forever") whenever `num` is less than 100?

```
cout << "enter number ";
cin >> num;
while (num < 100)
{   product = product * num;
    answer = answer + 1;
}
```

5.4 Write a loop that allows the user to enter a string, and that prints the first vowel that occurs in the string. Assume a boolean-valued function `IsVowel` exists that takes a string as a parameter and returns true if the string is vowel, otherwise returns false.

5.5 Write the function with the following specification.

```
string revstring(string s)
// pre: returns reverse of s, that is, "stab" for "bats"
```

Assuming `revstring` works, write a boolean-valued function `IsPalindrome` that returns true if a string is a **palindrome**, (i.e., is the same forwards as backwards like `"mom"` and `"racecar"`).

5.1.2 Loops and Mathematical Functions

The first computers were used almost exclusively as "number crunchers"—machines that solved numerical problems and equations. The very word "computer" formerly meant a person employed to perform such extensive calculations. For that reason, one of the first machines to do the job had the name ENIAC, for *E*lectronic *N*umerical *I*ntegrator *A*nd *C*omputer. This special-purpose computer eventually evolved into a more general machine called UNIVAC, for *Univ*ersal *A*utomatic *C*omputer.

Today the machines we now call "computers" are much more general-purpose, and many people find it difficult to imagine writing without using a word processor, movies without digital special effects, and banking without automatic tellers. All these applications require computers used in ways that at least on the surface don't involve numerical computations. Nevertheless, all information stored in today's computers is represented at some level by a number (even words are "converted" to 0's and 1's when stored in a computer's memory). **Numerical analysis** is a branch of computer science in which mathematical methods for solving many kinds of equations using computers are designed and developed. Although we won't delve deeply into this branch of computer science, we'll use some simple mathematical examples to study some broader concepts.

We'll investigate three mathematical functions: one to calculate the **factorial** of an integer, one to determine whether an integer is **prime,** and one to do **exponentiation** or raising a number to a power. These functions provide simple examples of loops and loop development, reinforce the concept of programmer-defined functions, and introduce functions to which we will return later.

5.1.3 Computing Factorials

The factorial function, usually denoted mathematically as $f(x) = x!$, is used in statistics, probability, and an area of computer science and mathematics called *combinatorics.* One definition of the function is

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n \tag{5.1}$$

so that $6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$. As a special case, by definition $0! = 1$. Program 5.2 implements and tests a function for computing factorials.

Program 5.2 fact.cpp

```
#include <iostream>
#include "prompt.h"
using namespace std;

// illustrates loop and integer overflow

long Factorial(int num);

int main()
{
    int highValue = PromptRange("enter max value for factorial",1,30);
    int current = 0;  // compute factorial of this value

    while (current <= highValue)
    {   cout << current << "! = " << Factorial(current) << endl;
        current += 1;
    }
    return 0;
}
```

```
long Factorial(int num)
// precondition: num >= 0
// postcondition returns num!
{
    long product = 1;
    int count = 0;

    while (count < num)          // invariant: product == count!
    {   count += 1;
        product *= count;
    }
    return product;
}
```

fact.cpp

In the function `Factorial` the variable `product` accumulates the result with the statement `product *= count;` this result is returned when the loop finishes executing. The values of the variables `product` and `count` change each time that the loop test is evaluated in computing 6!, as shown in Fig. 5.2.

OUTPUT

```
prompt> fact
enter max value for factorial between 1 and 30: 17
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 1932053504
14! = 1278945280
15! = 2004310016
16! = 2004189184
17! = -288522240
```

Each time that the loop test is evaluated, the value of the variable `product` is always equal to `(count)!` (that's `count` factorial), as shown. Since $0! = 1$ (by definition), this is true the first time the loop test is evaluated as well as after each iteration of the

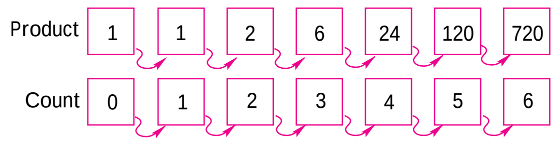


Figure 5.2 Relationship between variables `product` and `count` in *fact.cpp*.

loop body. A statement that is true each time a loop test is evaluated is called a **loop invariant**—the truth of the statement does not vary or change. Loop invariants can help us reason about the correctness of programs that use loops. Since `product == count!` is an invariant, and `product` is returned, we can reason that the `Factorial` function calculates the correct value if `count == num`. Since the loop test is false when the loop exits, and the logical negation of `count < num` is `count >= num` we're almost there. Since `count` is incremented by one, it cannot go past `num` without being equal to `num` first. Thus the loop test's negation, in conjunction with the invariant, help us reason about the correctness of the loop.

Conceptually, the function `Factorial` in Program 5.2 will always return the correct value. However, in practice the correct value may not be returned, as is evident from the foregoing run of the program. Note that 16! $<$ 15!; that 17! is a negative number; and that although 13! $= 13 \times 12!$, the value for 13! ends in a four while 12! ends in a zero. None of these results represents mathematical truth. Because integers stored in a computer have a largest value, it is possible for seemingly bizarre results to occur when this largest value is exceeded. Keep in mind that the limitation on integer values is one of many ways that a computer program can function exactly as it should (although not, perhaps, as intended), but produce unanticipated and often inexplicable results. I used `long` as the return type of `Factorial` and as the type of `product` to ensure that the function returns "correct" results through twelve factorial even on 16-bit machines.

Using the class `BigInt` instead of `int` or `long` allows calculations with arbitrarily large integers.[1] Details of the class `BigInt` can be found in Howto G, but you can program with them as though they were integers, that is use arithmetic operators, print them, and read them. Program 5.3, *bigfact.cpp*, shows how simple it is to use `BigInt` (you must use `#include"bigint.h"` when programming with `BigInt` values.)

Program 5.3 bigfact.cpp

```
#include <iostream>
#include "prompt.h"
#include "bigint.h"
using namespace std;
```

[1]The integers aren't really arbitrarily large, they're limited by the memory in the computer. In practice `BigInt` values are as big as you want; your programs will most likely run out of time in making calculations with them before running out of memory.

```
// illustrates loop and integer overflow

BigInt Factorial(int num);

int main()
{
    int highValue = PromptRange("enter max value for factorial",1,50);
    int current = 0;  // compute factorial of this value

    while (current <= highValue)
    {   cout << current << "! = " << Factorial(current) << endl;
        current += 1;
    }
    return 0;
}

BigInt Factorial(int num)
// precondition: num >= 0
// postcondition returns num!
{
    BigInt product = 1;
    int count = 0;

    while (count < num)          // invariant: product == count!
    {   count += 1;
        product *= count;
    }
    return product;
}
```

bigfact.cpp

OUTPUT

```
prompt> bigfact
enter max value for factorial between 1 and 50: 18

output of 0! to 11! is not shown here

12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
```

Unlike the results generated by Program 5.2, *fact.cpp*, the factorial calculations from *bigfact.cpp* are correct.

Cryptography and Computer Science

Before the 1970s, encryption techniques were largely based on sharing a private key that was used to encrypt messages. Both the sender and the receiver needed to have the private key. This was a potential security leak: how is the key transmitted from one person to another? In old movies couriers transported keys in briefcases strapped to their wrists. Apparently this method was used in real life as well.

In the mid-1970s several people developed **public-key** cryptography. The essence of these methods is that there are two keys: one private and one public. Everyone in the world has access to the public key and can use it to encrypt messages. Only the receiver of the message has the private key, and this key is required to decrypt the message. The keys are numbers and are calculated by choosing two large prime numbers, multiplying them together, and then doing a few other mathematical operations. The August 1977 “Mathematical Games” section of the magazine *Scientific American* explained this method of cryptography and had a challenge from the inventors of the method: Decrypt a message based on factoring the number called RSA-129 (it has 129 digits and is named for the inventors of the encryption method: Rivest, Shamir, and Adleman):

```
114,381,625,757,888,867,669,235,779,976,146,
612,010,218,296,721,242,362,562,561,842,935,
706,935,245,733,897,830,597,123,563,958,705,
058,989,075,147,599,290,026,879,543,541
```

The column claimed that it would take 40 quadrillion years to decrypt the message and offered $100.00 to the first person to do it. In 1994, more than 1600 computers around the world were put to work for eight months using new factoring methods to factor RSA-129. Coordinated by Arjen Lenstra, the computers used "wasted cycles"—time that the computers would have been otherwise idle—to factor RSA-129. The number was successfully factored, and the message from the *Scientific American* article decrypted. The message was `THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE`.

For an illuminating account of the method and history of public-key cryptography, and of a public-domain program called PGP that can be used for encrypting/decrypting, see [Gar95].

5.1.4 Computing Prime Numbers

Prime numbers used to be the domain of pure mathematicians specializing in number theory. Today, they play an increasingly important role in computer science applications. Current **encryption** techniques, used to encode data so that information cannot be read (electronically or visually), are largely based on efficient methods for determining whether a number is prime. Data encryption is a big business, with many ethical and privacy considerations. In addition, writing programs to determine whether numbers are prime is part of the rites of passage one traditionally undergoes in studying programming.

By definition, a number is prime if its only divisors are 1 and the number itself. For example, 5, 7, 53, and 97 are prime, but 91 is not prime, since it is divisible by 7 (and 13). The only even prime number is 2. By convention, 1 is not considered prime.

It seems that we'll need to check divisors of a number N to see whether the number is prime. We could naïvely check all numbers from 1 to N as potential divisors, checking the remainder each time. This method can be improved by checking only potential divisors less than the square root of a number. For example, to determine whether 119 is prime, we check divisors up to 11 (because $11^2 = 121 > 119$). Any number greater than 11 that divides 119 must have a corresponding factor less than 11, since factors come in pairs. Thus 7 and 17 are both factors of 119, but only one factor is needed to show that 119 is not prime (and the second factor is easily obtained by dividing by the first).

We need to be careful. We don't want to check that 1 is a divisor, since it divides every number evenly. We can also avoid testing even numbers as potential divisors, since 2 is the only even number that's prime. The approach used in the boolean-valued function `IsPrime` shown in Program 5.4 tests numbers less than or equal to 2 explicitly, avoids testing even numbers other than 2, then uses a loop to check all potential divisors less than the square root of `n`, the parameter of `IsPrime`.

Program 5.4 primes.cpp

```
#include <iostream>
#include <cmath>              // for sqrt
using namespace std;

// program to check for primeness
// Owen Astrachan, 4/1/99

bool IsPrime(int n);           // determines if n is prime

int main()
{
    int k,low,high;
    int numPrimes = 0;
    cout << "low number> ";
    cin >> low;

    cout << "high number> ";
    cin >> high;

    cout << "primes between " << low << " and " << high <<  endl;
    cout << "———————————-" << endl;

    k = low;
    while (k <= high)
    {   if (IsPrime(k))
        {   cout << k << endl;
```

```
                numPrimes += 1;
            }
            k += 1;
        }
        cout << "––––––-" << endl;
        cout << numPrimes << " primes found between " << low
             << " and " << high << endl;

        return 0;
}

bool IsPrime(int n)
// precondition: n >= 0
// postcondition: returns true if n is prime, else returns false
//                returns false if precondition is violated
{
    if (n < 2)                           // 1 and 0 aren't prime
    {   return false;                    // treat negative #'s as not prime
    }
    else if (2 == n)                     // 2 is only even prime number
    {   return true;
    }
    else if (n % 2 == 0)                 // even, can't be prime
    {   return false;
    }
    else                                  // number is odd
    {   int limit = int(sqrt(n) + 1);     // largest divisor to check
        int divisor = 3;                  // initialize to smallest divisor

        // invariant: n has no divisors in range [2..divisor)

        while (divisor <= limit)
        {   if (n % divisor == 0)         // n is divisible, not prime
            {   return false;
            }
            divisor += 2;                 // check next odd number
        }
        return true;                      // number must be prime
    }
}
```

primes.cpp

Each `return` statement in `IsPrime` exits the function. Flow of control continues with the statement that follows the call of `IsPrime`. In particular, the `return` statement in the while loop permits a kind of premature loop exit. As soon as a divisor is found, the function exits and returns `false`. If control reaches the `return` statement after the while loop, the loop test must be false; that is, `divisor > limit`. In this case `n` is prime.

OUTPUT

```
prompt> primes
low number> 100000
high number> 100100
primes between 100000 and 100100
-----------------------------------
100003
100019
100043
100049
100057
100069
-----------------
6 primes found between 100000 and 100100
```

Program Tip 5.1: When a `return` statement is executed, flow of control immediately leaves the function in which the `return` is located and continues with the statement that follows the function call. No other statements within the function are executed. One school of thought says that each function should have exactly one `return` statement. This is always possible, but it often requires the introduction of extra variables or more complicated code. You will find that judicious use of multiple `return`s within one function can make the function simpler to write and easier to reason about.

On some computers the assignment `int limit = sqrt(n) + 1` may cause a warning:

```
primes.cpp: In function ‘bool IsPrime(int)’:
primes.cpp:53: warning: initialization to ‘int’ from ‘double’
```

The value returned by `sqrt` is a `double`. Assigning a `double` to an `int` is not always possible, because the largest `double` value may be greater than the largest `int` value. Even though a program compiles, compiler warnings should not be ignored; they are often an indication that you have misused the language. In this case, you can avert the warning by explicitly converting the `double` value to an `int`. This is shown in *primes.cpp* in the statement assigning a value to `limit`:

```
int limit = int(sqrt(n) + 1);     // largest divisor to check
```

Using the type `int` like a function call explicitly converts the value `sqrt(n) + 1` into an integer. This is called a **type cast.** The cast prevents the warning, because you, the programmer, explicitly converted one type to another. We'll study casts in more detail in Section 6.3.6.[2]

The value `sqrt(n) + 1` is used instead of `sqrt(n)` because of the limited precision of floating-point numbers. For example, the square root of 49 might be calculated as 6.9999 rather than 7.0. In this case, the assignment `int limit = sqrt(49)` stores the value 6 in `limit`, because the `double` is truncated when it's assigned to an `int`. Adding 1 avoids this kind of problem.

5.1.5 Kinds of Loops

The loop in `main` of *primes.cpp* iterates exactly `high-low+1` times, so if `low` is 10 and `high` is 20, the loop executes 11 times, for `k` having values $10, 11, \ldots, 19, 20$. When the value of a simple arithmetic expression like `high-low+1` gives the number of loop iterations (and the value can be calculated before the loop executes the first time), the loop is called a **definite loop**. When the update of the expression used in the loop test is an increment by one, such as `k += 1` or a decrement by one, such as `k -= 1` in Program 5.1, *revstring.cpp*, the definite loop is often called a **counting loop**.

In contrast, the loop in the function `IsPrime` of *primes.cpp*, is not a definite loop. Although the maximum number of iterations can be calculated in advance[3] the loop exits as soon as a divisor is found. The early exit means that no simple expression determines the number of loop iterations. However, in some sense the loop is a kind of counting loop since the value of `divisor` is incremented by two for each iteration.

5.1.6 Efficiency Considerations

How important is it to check divisors up to $\sqrt{n}$ rather than n in determining whether a number is prime? People often suggest using $n/2$ rather than n. Is this better than $\sqrt{n}$? These questions are important in determining the efficiency or **complexity** of the algorithm used in `IsPrime`, but they don't affect the correctness of the algorithm. Consider that $\sqrt{50{,}000} = 223.6$, but that $50{,}000/2 = 25{,}000$. This difference means that using $n/2$ as the limit in `IsPrime` could result in approximately 12,388 more numbers being checked as potential divisors in determining that 49,999 is prime (it is). The extra number of divisors is 12,388 rather than 24,776 because only odd numbers are checked as potential divisors in the loop. I timed two versions of Program 5.4: one that used `limit = sqrt(n) + 1` and one that used `limit = n/2 + 1`. It took 1.44 seconds to determine that there are 5,133 primes between 1 and 50,000 when the square root limit was used, but 45.78 seconds when the limit based on half of `n` was used. Interestingly, even checking only divisors less than the square root of a number is much too slow for the encryption algorithms that are based on using large prime numbers. These encryption algorithms use pairs of large prime numbers, so they need to determine whether 200-digit numbers are prime. The square root of such a number has 100 digits. Testing 10^{100} numbers as potential divisors would require more time

[2]As we'll see in Section 6.3.6, the latest C++ standard has a casting operator `static_cast`, whose use is preferred to the style of cast we've shown here. Not all compilers support `static_cast`.

[3]The maximum number of iterations is roughly $\sqrt{n}/2$.

than the universe has been in existence. What makes the encryption algorithms feasible? Computer scientists and mathematicians developed efficient methods for determining whether a number is prime. These methods don't actually factor a number; they just yield a yes or no answer to the question "Is this number prime?" However, no one has developed an efficient algorithm for factoring numbers. The keys to the encryption methods used are (1) efficiently determining that a number is prime, and (2) difficulty in factoring the product of the two primes.

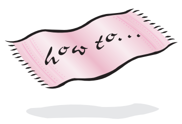
5.1.7 Exponentiation: A Case Study in Loop Development

We'll use the mathematical operation of raising a number to a power, called exponentiation, as an illustration of algorithm efficiency and of using invariants to develop a loop.

Today's businesses and governments rely increasingly on electronic messages and transactions. With powerful computers to assist in electronic spying, many worry that no message is safe from being stolen and deciphered. However, computer scientists have developed methods of data encryption that result in messages that are provably difficult to decrypt or decode.

These techniques of data encryption require that large prime numbers (approximately 150 digits) be manipulated by raising these numbers to large powers[4]. Data encryption is used to prevent people from "spying" on electronic information. For example, someone sending an electronic message from an office in Europe to an office in Canada might be worried that the message will be intercepted by electronic eavesdroppers. Instead of being sent as **plain text** (i.e., understandable by anyone), the message might be **encrypted** so that it cannot be intercepted and understood.

Efficient methods for computing x^n, the operation of **exponentiation,** are essential when both x and n are large. In C++, the exponentiation operation is not a built-in operator, as addition, subtraction, multiplication, division, and some others are (e.g., the % operator for remainder). The library of routines specified in the header file `<cmath>` does include an exponentiation routine called `pow` (see Table F.1), but it is useful for us to examine ways of implementing a function to perform exponentiation. Not only will doing so illuminate concepts of programming and C++; it is sometimes necessary to implement such a function when the one provided in the math library won't work, such as in raising a `BigInt` value to a power.



Exponentiation can be defined in at least three ways. The first method is the one you may be accustomed to.

$$a^n = \underbrace{a \times a \times \cdots \times a}_{n \text{ times}} \qquad (5.2)$$

An equivalent **inductive** or **recursive** definition follows.

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a \times a^{(n-1)} & \text{otherwise} \end{cases} \qquad (5.3)$$

[4]This is part of how *RSA* encryption works; the powers are computed modulo another number m so that the result is constrained to be between 0 and $m - 1$.

Table 5.1 Calculating 3^{16} Efficiently. The **Answer** column cannot be filled in until the **Depends On** column is filled in from the bottom to the top. x_i indicates a value to fill in.

Power	**Depends On**	**Answer**
$3^{16} = (3^8)^2$	$(x_4)^2$	43, 046, 721
$x_4 = 3^8 = (3^4)^2$	$(x_3)^2$	6, 561
$x_3 = 3^4 = (3^2)^2$	$(x_2)^2$	81
$x_2 = 3^2 = (3^1)^2$	$(x_1)^2$	9
$x_1 = 3^1$	*none*	3

Finally, it's possible to take advantage of properties of exponents such as $3^8 = 3^4 \times 3^4$ to define exponentiation.

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a^{n/2} \times a^{n/2} & \text{if } n \text{ is even} \\ a \times a^{n/2} \times a^{n/2} & \text{if } n \text{ is odd (note that } n/2 \text{ truncates to an integer)} \end{cases} \tag{5.4}$$

This last definition has the advantage that it may lead to fewer multiplications than the first two definitions when we develop a program to translate the definitions into code. For example, given the task of computing 3^{16}, you might break the problem down as shown in Table. 5.1, where 3^{16} is calculated by computing 3^8 and squaring the result. In turn, 3^8 is calculated by computing 3^4 and squaring it. This process of breaking down a number continues until the simple case of $3^1 = 3$ is reached. Note that only four multiplications are required—one for squaring each of the numbers: 3, 9, 81, 6561. This is many fewer multiplications than the 16 required by the naïve method used in the first definition of exponentiation above.

Definition 5.2 leads to the following relatively simple counting loop in the function `Power` for raising a number to an integer power.

```
double Power(double base,int expo)
// precondition:  expo >= 0
// postcondition: returns base^expo (base to the power expo)
{
   double result = 1.0;
   while (expo > 0)
   {   result *= base;
       expo -= 1;
   }
   return result;
}
```

The loop iterates exactly `expo` times so that calculating x^n requires n multiplications and n subtractions. We want to develop a similar function, one that is black-box equivalent to `Power`, but that uses fewer multiplications as with definition 5.4. We'll use a loop guard similar to the one above, but we'll use a loop invariant to help explain the loop and reason about its correctness. The invariant will also help you remember how to develop the code on your own. We'll start with the following code that accumulates the final answer in the variable `result`.

```
double Power(double base, int expo)
// precondition: expo >= 0
// postcondition: returns base^expo (base to the power expo)
{
   double result = 1.0;
   // invariant:  result * (base^expo) == answer
   while (expo > 0)
   {
   }
   return result;
}
```

Recall that a loop invariant is true each time the loop test is evaluated. In particular, it is true the first time the test is evaluated. The invariant is expressed as a comment:

$$\text{result} \times \text{base}^{\text{expo}} = \text{answer} \tag{5.5}$$

Since the initial value of `result` is 1.0, the invariant is true the first time the loop test is evaluated. Since `expo` is used in the loop test, the value of `expo` must change as the loop iterates. For the invariant to remain true, the values of either `result` or `base` must change as well. When the loop terminates, we'll want the value of `expo` to be zero. Since the invariant is true, this will guarantee that the correct answer is returned since $x^0 = 1$ for all x.

When the exponent is even, definition 5.4 dictates dividing the exponent by 2, that is taking advantage of the property that $3^{20} = 3^{10} \times 3^{10}$. If the exponent is divided in half then either `result` or `base` (or both) must change to establish the truth of the invariant. We'll use the following properties of even exponents.

$$a^b = a^{b/2} \times a^{b/2} = (a \times a)^{b/2} \tag{5.6}$$

Using this property, when we divide `expo` by 2 we'll square `base` so that the value of the expression in the invariant shown in Equation 5.5 remains the same.

$$\text{result} \times \text{base}^{\text{expo}} = \text{result} \times (\text{base} \times \text{base})^{\text{expo}/2} \tag{5.7}$$

This relationship leads to the following loop (the function header isn't duplicated).

```
double result = 1.0;
// invariant:  result * (base^expo) == answer
while (expo > 0)
{   if (expo % 2 == 0)          // exponent is even
    {   expo /= 2;
        base *= base;           // (a*a)^(b/2) == a^b
    }
    else  // must handle this case
}
return result;
```

The loop is almost done, but we must still deal with odd exponents. Definition 5.4 for odd exponents is similar to the case for even exponents, but an additional factor of base is involved, that is:

$$\text{result} \times \text{base}^{\text{expo}} = (\text{result} \times \text{base}) \times \text{base}^{\text{expo}/2} \times \text{base}^{\text{expo}/2} \tag{5.8}$$

The part of this expression involving expo/2 is identical to the expression used for even exponents. To incorporate the additional factor of base we'll multiply result by base. This re-establishes the invariant.

```
double result = 1.0;
// invariant:  result * (base^expo) == answer
while (expo > 0)
{   if (expo % 2 == 0)          // exponent is even
    {   expo /= 2;
        base *= base;           // (a*a)^(b/2) == a^b
    }
    else
    {   expo /= 2;
        result *= base;
        base *= base;
    }
}
return result;
```

Before we look at the code one final time, we'll review how the invariant helps reason about the correctness of the program.

1. The invariant is true each time the loop test is evaluated. In particular, it must be true the first and last times the test is evaluated.
2. When the loop finishes, the loop test must be false. We can use this, in conjunction with the truth of the invariant, to reason about a loop's correctness.

In the loop from Power, the value of expo will be zero when the loop exits. We can infer this because since the loop test is false, we know that expo <= 0. But expo can never be negative since it is only changed when it is divided by two. Since the invariant is true, and the value of expo is zero, we have the following:

$$\text{result} \times \text{base}^{\text{expo}} = \text{result} \times \text{base}^{0} = \text{result} = \text{final answer} \tag{5.9}$$

Since result is returned, we have "proved" that the function correctly satisfies its post-condition. Of course this is an informal proof, but hopefully it is effective in convincing you about the loop and the function.

Before you decide you're "done" in writing a function, class, or program, you should review the code. In the function Power the same statements appear in both the if and the else block. You should always **factor out** duplicated code by moving it before or after the if/else statement as appropriate. Here, we can factor out two statements, and leave an `if` without an `else`. To do this, we negated the original test used in the `if` so that now the code tests for odd exponents.

```
double Power(double base, int expo)
// precondition: expo >= 0
// postcondition: returns base^expo (base to the power expo)
{
   double result = 1.0;
   // invariant:  result * (base^expo) = answer
   while (expo > 0)
   {   if (expo % 2 != 0)      // exponent is odd
       {   result *= base;
       }
       expo /= 2;              // 4/2 == 2, 5/2 == 2
       base *= base;           // (a*a)^(b/2) == a^b
   }
   return result;
}
```

Program Tip 5.2: Invariants are useful in developing and documenting loops. You should try to include an invariant in every loop you write. At first this will seem difficult or useless. But what's obvious to you today won't be obvious to someone else, or to you tomorrow, so document, document, document your code.

Program Tip 5.3: Factor out common code. Don't be satisfied when your function or program works. Be sure that your code is easy to understand, is not uselessly redundant, and that code duplication is minimized.

Pause to Reflect

5.6 Assume that the factorial of a negative number is defined to be the factorial of the corresponding absolute value so that, for example, $(-5)! = 5! = 120$. Modify the function `Factorial` in Program 5.2 so that the correct value is returned for any value of `num`. Be sure to change the comments.

5.7 What value is returned by the call `Factorial(-7)` in the program *fact.cpp*, Program 5.2?

5.8 Write a function to calculate $x!!$ where $x!! = (x!)!$. For example, $3!! = 6! = 720$.

5.9 Generalizing the previous exercise, write a function with two parameters to calculate $x(!)^n$, where $x(!)^n = x \underbrace{!! \ldots !}_{n \text{ times}}$. Use `BigInts` for the calculations.

5.10 Here is another version of `Factorial`; this version is changed only slightly from that given in Program 5.2. Does this version pass a black-box test comparing it with the original? What is a good invariant for the loop?

```
int Factorial(int num)
{
    int product = 1;
    int count = 1;

    while (count <= num)
    {   product *= count;
        count += 1;
    }
    return product;
 }
```

5.11 If the statement `divisor += 2` is changed to `divisor += 1`, does the function `IsPrime` still work as intended?

5.12 What value is returned by the call `IsPrime(1)`? Is this what should be returned?

5.13 It is possible to write a loop without a return from the middle of the loop in the function `IsPrime`. The `while` loop can be replaced by the following:

```
while (divisor <= limit && n % divisor != 0)
{   divisor += 2;
}
```

What statement is needed after the loop to ensure that the correct value is returned?

5.14 What values does `expo` have each time the loop test is evaluated in the final version of the function `Power` if the original value is 1,024? If the original value is 1,000? (the last value is 0 in all cases).

5.15 Why is the invariant for the loop of `IsPrime` in *primes.cpp*, Program 5.4 true the first time the loop test is evaluated? Write an informal argument about the correctness of `IsPrime` using the invariant and the loop test together.

5.16 Before common code was factored out in the loop for calculating powers, the two statements below were part of the `else` clause.

```
result *= base;
base *= base;
```

Can the order of these statements be changed? Why?

5.17 Modify the function `Power` to work with negative exponents, where $a^{-n} = 1/a^n$.

5.1.8 Numbers Written in English

As another example of a loop we'll use `DigitToString` from *numtoeng.cpp*, Program 4.10, to convert a number to an English equivalent string formed from the digits. For example, 123 is represented by `"one two three"`, and 4017 is represented by `"four zero one seven"`.

We'll need a loop to do two things:

- Extract one digit at a time from the number
- Build up the string one word at a time

The modulus operator `%` makes it easy to determine the rightmost digit of any number. It's difficult to get the leftmost digit, because we don't know how many digits are in the number. To build the English equivalent, we'll have to build a `string` by concatenating each digit-string in the proper order. Each time a digit is peeled off the number, its corresponding string is concatenated to the front of the `string` being built.

Program 5.5 digits.cpp

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

// illustrates loops, convert a number to a string of English digits
// i.e., 1346 -> one three four six
// Owen Astrachan, 6/8/95

string DigitToString(int num);
string StringOut(long int number);

int main()
{
    long number;

    cout << "enter an integer: ";
    cin >> number;
    cout << StringOut(number) << endl;

    return 0;
}

string DigitToString(int num)
// precondition: 0 <= num < 10
// postcondition: returns english equivalent, e.g., 1->one,...9->nine
{
    if (0 == num)       return "zero";
    else if (1 == num)  return "one";
    else if (2 == num)  return "two";
    else if (3 == num)  return "three";
    else if (4 == num)  return "four";
    else if (5 == num)  return "five";
    else if (6 == num)  return "six";
    else if (7 == num)  return "seven";
    else if (8 == num)  return "eight";
    else if (9 == num)  return "nine";
    else return "?";
}

string StringOut(long number)
// precondition: 0 < number
// postcondition: returns string formed from digits written in English
//                e.g., 123 -> "one two three"
{
    string s = "";
    int digit;
    while (number != 0)
    {   digit = number % 10;
        s = DigitToString(digit) + " " + s;
        number /= 10;
    }
    return s;
}
```

digits.cpp

OUTPUT

```
prompt> digits
enter an integer: 9299338
nine two nine nine three three eight
prompt> digits
enter an integer: 401706
four zero one seven zero six
prompt> digits
enter an integer: -139
? ? ?
prompt> digits
enter an integer: 18005551212
eight two five six eight two zero two eight
prompt> digits
enter an integer: 8005551212
? ? ? ? ? ? ? ? zero
```

The first time the loop test is evaluated, `s` represents the empty string `""`: a `string` with no characters. The value of `digit` is undefined because no value has been assigned to `digit`. Since a space is always added after the digit string added to the front of `string s`, there is a space at the end of `s`. This space won't be "visible" if `s` is printed, unless another string is printed immediately after `s`. The space will be included in calculating the length of `s`, so `StringOut(111).length() == 12`, since the string is `"one one one "`.

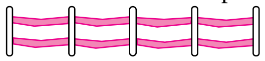
5.1.9 Fence Post Problems

The extra space after the last English digit of the `string` made by `StringOut` in Program 5.5, *digits.cpp*, is undesirable. Spaces should occur between each two digits rather than after each digit. A similar problem occurs with the following loop, intended to print the numbers 1 through 10 separated by commas: `1,2,3,4,5,6,7,8,9,10`. The loop doesn't work properly:

```
int num = 1;
while (num <= 10)
{   cout << num << ",";
    num += 1;
}
cout << endl;
```

The loop prints `1,2,3,4,5,6,7,8,9,10,` instead (note the trailing comma). The problem here is that the number of numbers is one more than the number of commas, just as the number of digits is one more than the number of spaces in the function *StringOut*.

This kind of problem is often called a **fence post** problem, because a fence (see picture below) has one more fence post than fence crosspieces. In our example, the numbers are the posts and the commas are the crosspieces.



The correct number of posts and crosspieces cannot be printed in a loop that outputs both fences and crosspieces, because the loop generates the same number of each. There are three alternatives: print the first fence post (number) before the loop; print the last post (number) after the loop; or guard the printing of the crosspiece inside the loop. The three approaches are coded as follows:

Program 5.6 threeloops.cpp

```
int n = 1;                   int n = 1;                  int n = 1;
cout << n;                   while (n < 10)              while (n <= 10)
n += 1;                      {   cout << n << ",";       {   cout << n;
while (n <= 10)                  n += 1;                     if (n < 10)
{   cout << "," << n;        }                                 cout << ",";
    n += 1;                  cout << n << endl;               n += 1;
}                                                         }
cout << endl;                                             cout << endl;
```

threeloops.cpp

In the solution on the left, the comma is printed before each number is printed in the loop. This requires an increment before the loop or a different initialization of `n`.

Printing the comma after each number requires printing the final number after the loop. This is shown in the code in the middle where the loop test is modified to use < instead of <=.

Both solutions share the problem of code duplication. In the code segment at the top left, `n` is incremented by one in two places. In the segment at the top right, there are two `cout << n` statements. Code duplication often causes maintenance problems, since changes must be made identically in more than one place. The solution on the right avoids the code duplication but mimics the loop test inside the loop, which is a slightly different kind of code duplication. Each of these solutions is an acceptable way to solve fence post problems.

Pause to Reflect

5.18 Write code that permits the user to enter the number of fence posts in a fence and that then "draws" a fence as shown in the following sample output:

OUTPUT

```
enter number of fence posts: 8
|---|---|---|---|---|---|---|
|---|---|---|---|---|---|---|
```

5.19 Alter the code in the function *StringOut* in Program 5.5, *digits.cpp*, so that spaces occur between each digit as opposed to after each digit.

5.20 Modify *StringOut* to generate a string that's backwards, for example, `"three two one"` for the number 123.

5.21 Write a function that returns the number of characters in an `int`, accounting for a minus sign for negative numbers. For example, `NumDigits(1234)` returns 4, and `NumDigits(-1234)` returns 5.

5.22 Write a loop that prints the numbers 1 through 100 with each group of 10 numbers starting on a new line. There should be a space between each of the numbers on a line:

```
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
...
91 92 93 94 95 96 97 98 99 100
```

You may find it useful to use the statement

```
if (num % 10 == 0)
{    cout << endl;
}
```

in the loop body.

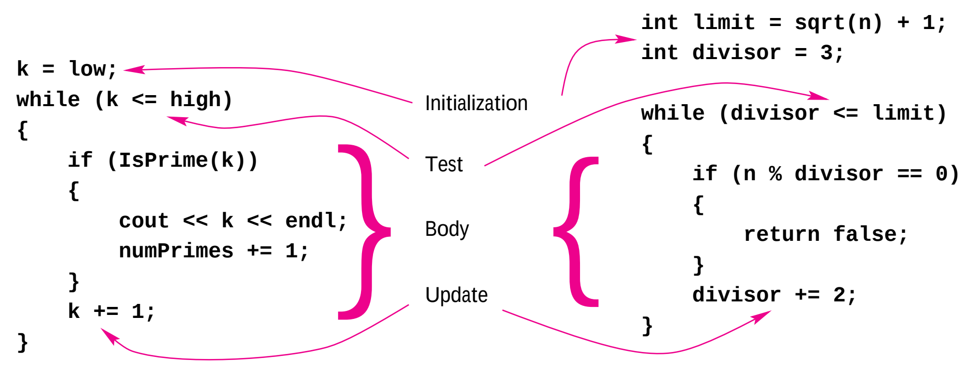
5.23 Write a loop using the operator `/=` that calculates how many times a number can be divided in half before 0 is reached. For example, 2 can be divided twice (attaining 1 then 0), 3 can be divided twice, 511 can be divided 9 times, and 512 can be divided 10 times. Use this loop to write a function `IntegerLog` that has two parameters, `number` and `n`, and returns how many times `number` can be divided by `n`.

5.2 Alternative Looping Statements

Writing loops can be difficult. It's not always easy to determine what the loop test should be, what statements belong in the loop body, and how variables should be initialized and updated before, in, and after a loop. Loops tend to have four sections:

- **Initialization:** This step occurs prior to the loop. Variables that need to be initialized are given values prior to the first time the loop test is evaluated.
- **Loop test:** The test determines whether the loop body will be executed. When the loop test is false, the loop body is not executed. If the loop test is always true, an infinite loop results, unless the loop is exited with a `return` statement, as used in `IsPrime` in *primes.cpp*, Program 5.4.
- **Loop body:** The statements that are executed each time the loop test evaluates to true.
- **Update:** The statements that affect values in the loop test. These statements ensure that the loop will eventually terminate. Values of variables in the loop test will be changed by the update statements.

These sections are diagrammed in Fig. 5.3 for the two loops in *primes.cpp*, Program 5.4



```
k = low;
while (k <= high)
{
    if (IsPrime(k))
    {
        cout << k << endl;
        numPrimes += 1;
    }
    k += 1;
}
```

```
int limit = sqrt(n) + 1;
int divisor = 3;

while (divisor <= limit)
{
    if (n % divisor == 0)
    {
        return false;
    }
    divisor += 2;
}
```

Figure 5.3 The four sections of a loop.

5.2.1 The `for` Loop

These loops are often written using an alternative looping construct to the `while` loop. The **for loop** is just a kind of shorthand, or **syntactic sugar,**[5] that can be used instead of a `while` loop. Anything written with one loop can be written with the other and vice versa.

The `for` loop offers some economy in terms of lines of code when compared with its `while` loop equivalent. The **initialization** statement is executed only once, before the evaluation of the test for the first time. The **test expression** is evaluated; if it is true, the loop body executes. After the last statement in the loop body is executed, the **update** statement executes. The test is then evaluated again, and the process continues (without initialization) until the test is false. Since all the information in a `for` loop appears at the beginning of the loop, it is often easier to understand than the corresponding `while` loop. The update statement should change values used in the test expression so that the loop makes progress toward termination.

Syntax: for loop

for (*initialization*; *test expression*; *update*)
{
    statement list;
}

Here is the `while` loop from `main` in *primes.cpp*, Program 5.4, with the corresponding `for` loop:

```
k = low;                           for(k=low; k <= high; k += 1)
while (k <= high)                  {   if (IsPrime(k))
{   if (IsPrime(k))                    {   cout << k << endl;
    {  cout << k << endl;                  numPrimes += 1;
       numPrimes += 1;                 }
    }                              }
    k += 1;
}
```

The parentheses following the `for` loop enclose three separate parts of a loop: initialization, test, and update. These parts are separated by semicolons as shown. Block statement delimiters enclose the body of the `for` loop just as they enclose the body of the `while` loop.

I adhere to a style of programming in which `for` loops are used only when a bound on the number of iterations can be simply calculated. Such loops are sometimes called **definite** loops. Typically, these loops are counting loops—loops that execute a sequence of statements a fixed number of times, as shown in the example above from *primes.cpp*. Many C++ programmers use `for` loops exclusively; the economy of code makes programs *appear* shorter. Choosing the style of loop to use should not be a major decision point in developing a program. Sticking with the style adopted in this book is one way of ensuring that little time is spent on deciding what kind of loop to use. As an example of when I choose *not* to use a `for` loop, a `while` loop from *digits.cpp*, Program 5.5, is shown in Program 5.7 on the left with the corresponding `for` loop on the right:

[5]The term **syntactic sugar** is used for constructs that don't have a new meaning but are more aesthetically pleasing in some way. Often this means "easier for a human reader to understand."

Program 5.7 digitloops.cpp

```
while (number != 0)                      for(; number != 0; number /= 10)
{  digit = number % 10;                  {  s = DigitToString (digit) + " " + s;
   s = DigitToString(digit) + " " + s;   }
   number /= 10;
}
```

digitloops.cpp

This is *not* a counting loop. The number of times the loop body is executed depends on how many times `number` can be divided by ten.[6] This example shows that the initialization part of a `for` loop can be omitted. The other parts of a `for` loop can be omitted too, but omitting the test part results in an infinite loop.

5.2.2 The Operators ++ and −−

Counting loops often require statements such as `k += 1`. Because incrementing by one is such a common operation, C++ includes an operator that can be used to increment by one in place of `+= 1`. The statement

```
k++;
```

can be used in place of `k += 1`. Similarly, the statement `k--` can be used in place of `k -= 1` to decrement a value by one. The operator `++` is the **postincrement** operator, and the operator `--` is the **postdecrement** operator. In all the code in this book, the expression `x++` is used only as shorthand for `x += 1`. Similarly, `x--` is used only as shorthand for `x -= 1`. If you read other books on C++, you may find these operators used as parts of other expressions. For example, the statement `x = z + y++` is legal in C++. This statement stores `z + y` in `x`, and then increments the value of `y` by one. Don't try to use the operator this way—it will invariably get you into trouble. Instead, use `++` and `--` only as abbreviations as already described. When used in this way, the statements below on the left affect `x` the same way: its value is incremented by one. Similarly, the statements on the right decrement `x` by one.

```
x += 1;                         x -= 1;
x++;                            x--;
++x;                            --x;
```

I don't use the **preincrement operator** `++x` or the **predecrement operator** `--x` in this book. When used in expressions like `x = z + ++y`, the value of `y` is incremented first, then the value of `z + y` is stored in `x`. Since I don't use `++` and `--` except as abbreviations for `+= 1` and `-= 1`, I use only the postincrement and postdecrement operators.

An example of a counting `for` loop using the postincrement operator follows; this is the `while` loop from `main` of *primes.cpp*, Program 5.4:

[6]Although this number of iterations can be calculated using logarithms, this isn't done in this loop.

```
for(k = low; k <= high; k++)
{   if (IsPrime(k))
   {   cout << k << endl;
       numPrimes++;
   }
}
```

Pause to Reflect

5.24 The function `Factorial` in *fact.cpp* Program 5.2, uses a `while` loop to calculate the factorial of a number. Rewrite the function so that a `for` loop is used instead.

5.25 Write a `while` loop equivalent to the following `for` loop:

```
double total = 0.0;
double val;
for(val = 1.0; val < 10000; val *= 1.5)
{   total += val;
}
```

5.26 Write a `for` loop equivalent of the following `while` loop:

```
int k = 1;
int sum = 0;
while (k <= num)
{   sum += k;
    k += 2;
}
```

5.27 What is printed by the following `for` loop?

```
int k;
for(k=1024; k >= 0 ;k/=2)
{   cout << k << endl;
}
```

5.2.3 The do-while Loop

Many programs prompt for an input value within a range. For example, Program 4.10, *numtoeng.cpp*, prompts for an `int` between 0 and 100. The `PromptRange` functions declared in `"prompt.h"` ensure that input is within a range specified by the programmer. Think for a moment about how to write a loop that continually reprompts if input is not within a specific range. Since you must enter a value before any test can determine whether the value is valid, using a `while` loop leads to a fence post problem. Instead, a **do-while** loop can be used. The `do-while` loop works similarly to a `while` loop, but the loop test occurs at the end of the loop rather than at the beginning. This means that the body of a `do-while` loop is executed at least once. In contrast, a `while` loop does not iterate at all if the loop test is false the first time it is evaluated. Here is the body of one of the `PromptRange` functions (from *prompt.cpp*):

```
int PromptRange(string prompt,int low, int high)
// pre: low <= high
// post: returns a value between low and high (inclusive)
{
   int value;
   do
   {   cout << prompt << " between ";
       cout << low << " and " << high << ": ";
       cin >> value;
   } while (value < low || high < value);

   return value;
}
```

Note that the output statements for the prompt are executed prior to the input statement. If the value entered is not valid, the loop continues to execute until a valid value is entered.

5.2.4 Pseudo-Infinite Loops

Because of errors in design, loops sometimes execute forever. It's fairly common to forget to increment a counter when writing a `while` loop. This is a good reason to use a `for` loop—it's harder to forget the update statement in a `for` loop.

Sometimes, however, it's useful to write seemingly infinite loops with an exit condition from within the loop body. We'll use this style of loop with an exit only in situations that would cause code to be duplicated otherwise. Consider the following loop, which sums user-entered values until the user enters zero:

```
int sum = 0;
int number;
cin >> number;
while (number != 0)
{   sum += number;
    cin >> number;
}
cout << "total = " << sum << endl;
```

To evaluate the test `while (number != 0)`, the variable `number` is given a value before the test is evaluated for the first time as well as each time the loop body is executed. Reading an initial value so that the loop test can be evaluated the first time is called **priming** the loop. A word is read again within the loop body before the next evaluation. Eric Roberts, author of *The Art and Science of C*, calls these "loop-and-a-half" loops [Rob95]. Studies show that loop-and-a-half[7] loops are easier for students to write as infinite loops with an exit.

[7]It would be nice to say that four out of five programmers surveyed prefer `while (true)` with `break` loops. Studies do indicate that students find it easier to write code using this kind of loop than using a primed `while` loop.

The following loop avoids duplicating the code that extracts a value for `number` from `cin`:

```
while (true)            // until break from within loop
{   cin >> number;
    if (number == 0)
    {   break;          // OUT OF LOOP
    }
    sum += number;
}
cout << "total = " << sum << endl;
```

Since the loop test is always true, the loop appears to be an infinite loop. There is no way for the test to become false. The **break** statement in the loop causes an abrupt change in the flow of control. When executed, a `break` causes execution to break out of the innermost loop in which the `break` occurs. In the example here, execution continues with the output statement `cout << "total = ..."` when the `break` is executed. As an alternative to `while(true)`, the loop test `for(;;)` is a special C++ idiom that also means "execute forever." I don't use this style of infinite loop since its purpose doesn't seem as clear as the `while(true)` loop.

It is easy to carry this style of writing loops to extremes and write only infinite loops with break statements. You should try to write loops with explicit loop tests and use `while(true)` loops only for loop-and-a-half problems.

Program Tip 5.4: The `break` statement causes termination of the innermost loop in which it occurs. Control passes to the next statement after the innermost loop. Use the `break` statement judiciously in situations where code would be duplicated otherwise. As we'll see in later chapters, loop tests often provide meaningful clues when it becomes necessary to reason about how a loop works and whether or when the loop terminates. A test of `true` doesn't provide many clues. However, used properly, infinite loops avoid code duplication and thus lead to programs that are easier to maintain.

Some programmers find it easier to understand the logic of the following loop than that of the loop used in `PromptRange` shown previously:

```
while (true)
{   cout << prompt << " between ";
    cout << low << " and " << high << ": ";
    cin >> value;
    if (low <= value && value <= high) return value;
}
```

The `return` statement exits the function (and the loop) when the user-entered value is within the specified range. Sometimes it's easier to develop the logic for loop termination, as shown above, than for loop continuation, as shown in the function `PromptRange`. De Morgan's law from Section 4.7 can help in converting logical expressions for continuation into expressions for termination since one is typically the logical negation of the other.

5.2.5 Choosing a Looping Statement

The `while` loop is the kind of loop to use in most situations. For writing definite loops, a `for` loop may be appropriate. For writing loops that must iterate once, a `do-while` loop may be appropriate. Given that there are three different kinds of loops, it's natural to wonder whether there are rules that can make the "correct" choice of what kind of loop to apply easier to determine. Since any loop can be made to do the work of any other by using appropriate statements, we won't worry too much about this kind of decision. In summary, however, the following guidelines may prove helpful:

- The `while` loop is a general-purpose loop. The test is evaluated before the loop body, so the loop body may never execute.
- The `for` loop is best for definite loops—loops in which the number of iterations is known before loop entry.
- The `do-while` loop is appropriate for loops that must execute at least once, because the test is evaluated after the loop body.
- Infinite loops, with a `break` (or `return` from function) statement, are often useful alternatives, especially when loop priming is necessary or when it's difficult to develop the logic used in the loop test.

In all three types of loop the braces {} that surround the loop body are not required by the compiler if the loop body is a single statement. However, the style guidelines for code in this book require the bodies of loops and `if`/`else` statements to be enclosed in braces, even if they consist of single comments.

5.2.6 Nested Loops

When one loop occurs in the body of another loop, the loops together are called **nested loops**. In *primes.cpp*, Program 5.4, there is a "virtual nested loop," because the loop in the function `IsPrime` is executed repeatedly by the call from the loop in `main`.

An example adapted from [KR96] shows how nested loops can be used to print a table of wind chill values. The effective temperature is significantly decreased when the wind speed is high. For example, a 20 mile-per-hour wind on a 50-degree day reduces the temperature to an equivalent wind chill index of 32 degrees. The desired output is a table of wind speed and temperature, with the wind chill index temperature given as follows:

OUTPUT

```
prompt> windchill
deg. F:    50   40   30   20   10    0  -10  -20  -30  -40

0 mph:     50   40   30   20   10    0  -10  -20  -30  -40
5 mph:     47   37   26   16    5   -4  -15  -25  -36  -47
10 mph:    40   28   15    3   -9  -21  -33  -46  -58  -70
15 mph:    35   22    8   -4  -18  -31  -45  -58  -72  -85
20 mph:    32   17    3  -10  -24  -39  -53  -67  -82  -96
25 mph:    29   14    0  -14  -29  -44  -59  -74  -89 -104
30 mph:    28   12   -2  -17  -33  -48  -63  -79  -94 -109
35 mph:    26   11   -4  -20  -35  -51  -67  -82  -98 -113
40 mph:    25    9   -5  -21  -37  -53  -69  -85 -101 -116
45 mph:    25    9   -6  -22  -38  -54  -70  -86 -102 -118
50 mph:    25    9   -7  -23  -39  -55  -71  -87 -103 -119
```

Because the table must be printed one row at a time, a first cut at the code is row-oriented, with one row for each wind speed between 0 and 50 miles per hour:

```
for(windspeed=0; windspeed <= 50; windspeed += 5)
    print a row of temperatures;
```

Printing a row also requires a loop, and this leads to the nested loops shown in *windchill.cpp*, Program 5.8. Each wind chill temperature is printed by the **inner loop,** in which temperature varies from 50 down to −40 degrees; the inner loop prints a complete row of the table. The inner loop executes completely before one iteration of the **outer loop** has finished.

Program 5.8 windchill.cpp

```
#include <iostream>
#include <iomanip>       // for setw
#include <cmath>         // for sqrt
using namespace std;

// Owen Astrachan
// nested loops to print wind-chill chart
//
// idea: Programming with Class by Kamin and Reingold, McGraw-Hill
// formula for wind-chill from
// UMAP Module 658, COMAP, Inc., Lexington, MA 1984, Bosch and Cobb

double WindChill(double temperature, double windSpeed);

int main()
```

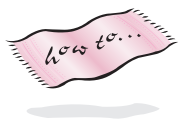
```
{
    const int WIDTH = 5;
    const int MIN_TEMP = −40;
    const int MAX_TEMP = 50;
    const string LABEL = "deg. F: ";
    int temp,wind;

    // print column headings

    cout << LABEL;
    for(temp = MAX_TEMP; temp >= MIN_TEMP; temp −=10)
    {   cout << setw(WIDTH) << temp;
    }
    cout << endl << endl;

    // print table of wind chill temperatures

    for(wind = 0; wind <= MAX_TEMP; wind += 5) // row heading
    {   cout  << wind << " mph:\t";

        for (temp = MAX_TEMP; temp >= MIN_TEMP; temp −= 10)  // print the row
        {   cout << setw(WIDTH) << int(WindChill(temp,wind));
        }
        cout << endl;
    }
    return 0;
}

double WindChill(double temperature, double windSpeed)
// precondition: temperature in degrees Fahrenheit
// postcondition: returns wind-chill index/comparable temperature
{
    if (windSpeed <= 4)          // low wind, temperature unaltered
    {   return temperature;
    }
    else if (windSpeed <= 45)    // high wind
    {   return
           91.4 − (10.45 + 6.69∗sqrt(windSpeed) − 0.447 ∗ windSpeed) ∗
           (91.4 − temperature)/22.0;
    }
    else
    {   return (1.6 ∗ temperature − 55.0);
    }
}
```

windchill.cpp

Because the function `WindChill` returns a `double` value and there is no reason to print several numbers after a decimal point in the table, the value returned by the `WindChill` function is stored in an `int` variable. The value is converted to an `int` using the expression `int(WindChill(temp,wind))` just as the value returned by the function `sqrt` was converted to an `int` in *primes.cpp*, Program 5.4. To make each column of the table line up properly, a stream manipulator `setw` for the input stream `cout` is used. The argument to `setw` specifies a **field width** used to print the next value. Printing a number like 27 in a field width of five requires three extra spaces in addition to the two characters of `27` to pad the output to five characters. If the output occupies three spaces (e.g., the number 123 or the string `"cat"`), then two literal blanks `'  '` will pad the output to five spaces. If the value being printed requires more than five spaces (e.g., for the number 123456), the entire value is still printed. You don't need `setw`; it's possible to print the right number of spaces by testing the value being printed as follows and padding with spaces as shown below, but using `setw` is much simpler.

```
if (num < 10)
{   cout << "  ";     // two spaces
}
else if (num < 100)
{   cout << " ";      // one space
}
cout << num;
```

Program output should be easy to read, but you should not concentrate on well-formatted output when first implementing a program. Information on `setw` and other functions that help in formatting output is in Howto B.



Sometimes it is useful to use the value of the outer loop to control how many times the inner loop iterates. This is shown in *multiply.cpp*, Program 5.9, which prints the lower half of a multiplication table (the upper half is the same, because multiplication is commutative: $2 \times 5 = 5 \times 2$). Both loops are counting loops. The outer loop, whose loop control variable is `j`, determines how many rows appear in the output. The statement `cout << endl` is executed once each time the body of the outer loop is executed. The number of iterations of the inner loop is determined by the value of `j`. As can be seen in the output, the number of entries in each row increases by 1 in each successive row. When `j` is one, there is one number, `1`, in the first row. When `j` is three, there are three numbers, `3 6 9`, in the third row. The `width` member function ensures that three-digit numbers and two-digit numbers line up properly in columns.

Program 5.9 multiply.cpp

```
#include <iostream>
#include <iomanip>   // for setw
#include "prompt.h"
using namespace std;

// simple illustration of nested loops

int main()
{
    int j,k;
    int limit = PromptRange("number for multiply table",2,15);

    for(j=1; j <= limit; j++)
    {   for(k=1; k <= j; k++)
        {   cout << setw(3) << k*j << " ";
        }
```

```
            cout << endl;
        }
        return 0;
    }
```

multiply.cpp

OUTPUT

```
prompt> multiply
number for multiply table between 2 and 15: 5
  1
  2   4
  3   6   9
  4   8  12  16
  5  10  15  20  25

prompt> multiply
number for multiply table between 2 and 15: 10
  1
  2   4
  3   6   9
  4   8  12  16
  5  10  15  20  25
  6  12  18  24  30  36
  7  14  21  28  35  42  49
  8  16  24  32  40  48  56  64
  9  18  27  36  45  54  63  72  81
 10  20  30  40  50  60  70  80  90 100
```

If a `break` statement is inserted as the last statement of the inner loop, immediately following `cout << setw(3) << k*j << " "`, the output changes:

OUTPUT

```
prompt> multiply
number for multiply table between 2 and 15: 4
1
2
3
4
```

Note that the outer loop is *not* exited early. The `break` statement causes the inner loop (in which the loop control variable is `k`) to exit before the loop test `k <= j` becomes false. This means that the inner loop executes exactly once.

You should think very carefully when you decide that nested loops are necessary, especially if you're using `while` loops. Nested loops are often necessary when data are printed or processed in a tabular format, but it is often possible to use a single loop with an `if` statement in the loop body, and one loop is usually easier to code properly than two nested loops are.

Program Tip 5.5: Coding is often easier if you move the inner loop of a nested loop into a separate function, and then call the function. It's often easier to test a function than to test a loop, and keeping the inner loop in a separate function helps in developing correct programs.

5.2.7 Defining Constants

In *windchill.cpp*, Program 5.8 several **constant** identifiers are defined.

```
const int WIDTH = 5;
const int MIN_TEMP = -40;
const int MAX_TEMP = 50;
```

The type modifier **const** means that `MIN_TEMP` is a constant. Because it is constant, `MIN_TEMP` cannot be assigned a new value or changed in any way. For example, if the line

```
MIN_TEMP = -80;
```

is added immediately after the definition of `MIN_TEMP`, one compiler generates the error message below.[8]

```
Error : not an lvalue
windchill.cpp line 19   MIN_TEMP = -80;
```

In general, it is good programming practice to use constants to represent values that do not change during the execution of a program. Some examples of constant definitions are the following:

```
const double PI = 3.14159265;
const double INCHES_PER_CM = 0.39370;
const int January = 1;
const string cpp = "C++";
```

[8]An *lvalue* is an object to which a value can be assigned; the "l" is for left, since assignment changes the variable on the left.

Using named constants not only improves the readability of a program; it permits edit changes in a program to be localized in one place. For example, if you need a more precise value of π of 3.1415926535897, only one constant is changed (and the program recompiled). Mnemonic names, or names that indicate the purpose they serve, also provide meaning and make it easier to read and understand code. Using the constant `January` instead of 1 in a calendar-making program can make the code much easier to follow. It is a common convention for constant identifiers to consist of all capital letters and to use underscores to separate different words.

Syntax: const value

`const` *type identifier* = *value*;

Using constants also protects against inadvertent modification of a variable. The compiler can be an important tool in developing code if you use language features like `const` appropriately.

Pause to Reflect

5.28 Write a loop that accepts input from the user until the number zero is entered. The output should be the number of positive numbers entered and the number of negative numbers entered.

5.29 There is a fence post problem in *multiply.cpp*: a space is printed after every number rather than between numbers. Modify the loop so that no space is printed after the last number in a row (*Hint:* it's possible to do this by modifying how `setw` is used).

5.30 Write nested loops to print (a) the pattern of stars on the left and (b) the pattern of stars on the right. The number of rows should be entered by the user; there are k stars in row k.

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

5.31 Write appropriate constant definitions to represent the number of feet in a mile (5,280); the number of ounces in a pound (16); the mathematical constant e (2.71828); the number of grams in a pound (453.59); and the number foot-pounds in an erg (1.356×10^7).

5.3 Variable Scope

In Section 3.1.2 we discussed where variables are defined, and we showed that it is possible to define variables anywhere, not just immediately following a curly brace {. You need to be aware of how the location of a variable's definition affects the use of the variable. For example, the variable `numPrimes`, defined in `main` of *primes.cpp*, Program 5.4, is not accessible from the function `IsPrime`. A variable defined within a function is **local** to the function and cannot be accessed from another function. Parameters provide a mechanism for passing values from one function to another.

Similarly, a variable defined between two curly braces { } is accessible only within the curly braces. To be more precise, a variable name can be used only from the point at which it is defined to the first right curly brace }. For example, consider the following fragment from the function `IsPrime`. The variables `limit` and `divisor` are accessible only within the `else` block in which they are defined. The added comment after the else block indicates that these variables cannot be accessed at that point.

```
else                             // number is odd
{
    int limit = int(sqrt(n) + 1);
    int divisor = 3;  // smallest divisor

    while (divisor <= limit)
    {   if (n % divisor == 0)  // n is divisible, not prime
        {   return false;
        }
        divisor += 2;          // check next odd number
    }
    return true;               // number must be prime
}

// comment added: limit and divisor NOT defined here
```

The following code fragment shows a variable `count` that can only be accessed in the bottom "half" of a loop:

```
while(total <= limit)
{   // count NOT accessible here

    int count = 0;

    // count IS accessible here
}
// count NOT accessible here
```

The variable `count` is accessible only from within the loop, and only from its definition to the bottom of the loop. The part of a program in which a variable name is accessible is called the variable name's **scope**.

You should be careful when defining variables in loop bodies (or `if`/`else` blocks), because these variables will not be accessible outside the loop body. In particular, be careful of `for` loops written as follows:

```
for(int k=0; k < 10; k++)
{   // loop body
}
```

The variable `k` is not, strictly speaking, defined within the curly braces that delimit the body of the loop. Nevertheless, the scope of `k` is local to the loop; `k` cannot be accessed after the loop. Not all compilers support this kind of scoping with `for` loops, but according to the C++ standard the scoping should be supported. It is common to need to access the value of a loop index variable (`k` in the example above) after the loop has finished. In such a case, the loop index cannot be local to the loop.

5.4 Using Classes

In addition to built-in C++ types like `int` and `double`, we've made extensive use of the class `string` in the programs we've studied so far. In Section 5.1.3 we showed how the `BigInt` class was useful for representing integers without the limits on values inherent in using the `int` type. In this section we extend our programming toolkit by looking briefly at two classes: the `Date` for representing calendar dates and the class `Dice` for simulating the kind of dice used in board games. We'll look at these classes as **client** programmers or users of the classes rather than implementers. In the next chapter we'll look more closely at how classes are implemented, but here we're more interested in extending the kinds of programs we can write by using classes, rather than studying the classes themselves.

5.4.1 The `Date` Class

In general, manipulating and understanding dates and calendars is an integral part of many software products. The so-called *year 2000 problem* has cost companies billions of dollars as they try to cope with software written when memory and disk space were expensive; the software typically uses two digits to represent a year, that is 99 represents 1999. The year 2000 will cause problems with much of this software because, for example, a credit card issued in 1998 that expires in two years might be stored in software as expiring in the year 100 (two years after the year 98). Careless programming and design can lead to serious problems with such products. In [Neu95] several potential problems with software that manipulates dates and times are illustrated:

- With COBOL (COmmon Business-Oriented Language), a programming language used extensively in business and finance, most software allocates only two digits for the year part of a date. This will undoubtedly cause problems in switching from December 31, 1999, to January 1, 2000.
- Early releases of the spreadsheet program Lotus 1-2-3 treated 2000 as a nonleap year and 1900 as a leap year when, in fact, the opposite is the case. Later versions of the software corrected the problem for the year 2000, but not for 1900, which remains a leap year according to the software.
- A Washington, D.C., hospital computer crashed on September 19, 1989, precisely 32,768 days after January 1, 1900. Note that 32,767 is the largest integer representable by an `int` on typical microcomputers.

We'll examine a class that represents calendar dates for any month in any year after October, 1752.[9] Some of the tools for implementing a calendar date class have been developed already in previous programs: determining the number of days in a month and determining when a year is a leap year.

Rather than use these tools to develop code that calculates the day of the week, we'll use a class `Date`, accessible using the include file `"date.h"`. In making a calendar, not all of the member functions of the `Date` class will be used. (Full details of the class can be found in Howto G.) Instead, we'll rely on a simple example program to understand how to use some of the member functions of the class `Date`.

Program 5.10 usedate.cpp

```
#include <iostream>
#include "date.h"
using namespace std;

// show Date member functions

int main()
{
    Date today;
    Date birthDay(7,4,1776);
    Date million(1000000L);
    Date badDate(3,38,1999);
    Date y2k(1,1,2000);

    cout << "today   \t: "  << today    << endl;
    cout << "US bday \t: "  << birthDay << endl;
    cout << "million \t: "  << million  << endl;
    cout << "bad date \t: " << badDate  << endl << endl;

    cout << y2k << " is a " << y2k.DayName() << endl << endl;

    Date one = million - 999999L;
    Date birthDay2000(birthDay.Month(), birthDay.Day(), 2000);
    today++;

    cout << "day one \t: "   << one << " on a " << one.DayName() << endl;
    cout << "bday2K  \t: "   << birthDay2000 << endl;
    cout << "tomorrow \t: "  << today << endl;

    return 0;
}
```

usedate.cpp

[9]The calendar used in the United States is the *Gregorian* calendar, which went into effect in 1582, but not in the English-speaking world until 1752. Several countries did not adopt this calendar until the 1900s, but it is adopted almost universally today. In-depth and interesting information about calendars can be found in [DR90, RDC93].

In reading the output below it might help to know that I ran the program on March 15, 1999. Think about what appears on each line of the output and how the `Date` class works.

OUTPUT

```
prompt> usedate
today           : March 15 1999
US bday         : July 4 1776
million         : November 28 2738
bad date        : March 1 1999

January 1 2000 is a Saturday

day one         : January 1 1 on a Monday
bday2K          : July 4 2000
tomorrow        : March 16 1999
```

Constructors and Initialization. The technical word that describes object initialization and definition is **construction.** Construction initializes the state of an object. For programmer-defined classes like `Date`, a special member function, called a **constructor,** performs this initialization. The first line of output from *usedate.cpp* will differ depending on the day the program is run. This is because the variable `today`, defined using the parameterless or **default** constructor, constructs a variable with “today’s date” according to the documentation in *date.h,* Program G.2. The variable `birthDay` is constructed using the three-parameter constructor. According to the documentation in *date.h* the parameters specify the month, day, and year of a `Date` object. The variable `million` is constructed using the single-parameter constructor. The documentation in *date.h* indicates that the value of the parameter specifies the absolute number of days from January 1, A.D. 1; one million days from this date is November 28, 2738.[10] Finally, the variable `badDate` is constructed with an invalid date in March; the invalid date is converted to March 1 (as described in the beginning of the header file.) Invalid months (i.e., outside the range 1–12) are converted to January.

Classes often have more than one constructor, especially when there is more than one way to specify the value of an object. The compiler can determine which constructor to use since the parameter lists are different.

[10]In the constant value 1000000L, the L is used to indicate that this is a `long int` value. On 32-bit machines the L isn’t necessary, but it is needed on 16-bit machines where the largest `int` value is 32,767.

Other `Date` Member Functions. Based on the output of *usedate.cpp* you may be able to determine that the `Date` member function `DayName()` returns the day of the week on which a date occurs. You can check a calendar to see that New Year's day in 2000 is a Saturday (which makes it convenient to celebrate on Friday night!) The functions `Month()` and `Day()` return the number of the month (1 . . . 12) and day, respectively, for a given date. These return `int` values, as you might have determined by the similarity of the construction of `birthDay2000` to `birthDay`.

It's also possible to perform arithmetic with `Date` objects. The variable `one` is constructed by subtracting a (long) integer value from the `Date` object `million`. This yields another date, in the same way that the value of `today - 1` is a `Date` representing yesterday. The statement `today++` changes `today` to represent the next day, or tomorrow. Of course it's confusing that the value of `today` becomes tomorrow after the statement executes.

You can compare dates using the relational operators, such as `<`, `<=`, and others. For complete information, see the header file `"date.h"` and the exercises at the end of this chapter.

Pause to Reflect

5.32 How would you use a `Date` variable to determine on what day of the week you were born?

5.33 How would you use the `Date` class to determine how many days you've been alive (hint: subtract two `Date` objects)?

5.34 Using one `Date` variable and the member function `DaysIn()` (that returns the number of days in the month, see `date.h`) write the boolean-valued function `IsLeapYear` as specified, in *isleap.cpp*, Program 4.8.

5.35 If the one-millionth day is November 28, 2738 (see *usedate.cpp*), do we need to worry that the `Date` class is not robust and might cause problems when the absolute number of days since 1 A.D. exceeds the largest value of a `long`?

5.36 In Canada and Europe dates are usually specified by giving the day first rather than the month. In the United States, 4/8/2000 means April 8, 2000. The same date means August 4, 2000 in Canada. Is it possible to write a program using the `Date` class for dates in Canada? How?

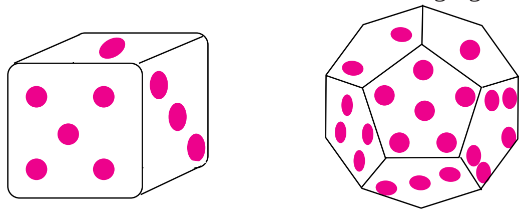
5.37 Write a function that determines and returns the Date on which Thanksgiving (a U.S. holiday) occurs in any year. Thanksgiving is the fourth Thursday in November. Use the following header.

```
Date Thanksgiving(int year)
// post: returns the Date for Thanksgiving in year
```

5.38 Many people prefer Fridays to Mondays. Write a function that prints all the months in a given year that have more Fridays than Mondays.

5.4.2 The Dice Class

In this section you'll learn about a programmer-defined class, named `Dice`, that permits the computer to simulate the kind of dice used in board games. The class simulates dice with any number of sides, not just common six-sided dice. It's even possible to have one-sided dice and million-sided dice, both of which are easy to simulate but hard to carve. Six- and twelve-sided dice are shown in the following figure.



The class `Dice` is very general and permits simulation of an N-sided die for any N.

These simulated dice, and the computer-generated random numbers on which they are based, are part of an application area of computer science called **simulation.** Simulations model real-world phenomena using a computer, which becomes a virtual laboratory for experimenting with models of physical systems without the expense of building the systems. Computer-based simulations are used to design planes, trains, and automobiles; to predict the weather; and to build and design computers and programs. We'll study simulation in more detail in the later chapters, but we'll use the `Dice`[11] class to study program and class construction.

To use the `Dice` class in a program you must include `"dice.h"` just as you must include `"date.h"` to use the `Date` class and `<string>` to use the `string` class. (The header file for the `Dice` class is in Howto G.) Program 5.11 is a simple program showing all the `Dice` member functions.

Program 5.11 roll.cpp

```
#include <iostream>
using namespace std;
#include "dice.h"

// simple program illustrating use of Dice class
// roll two dice, print results, Owen Astrachan, 3/31/99

int main()
{
    Dice cube(6);              // six-sided die
    Dice dodeca(12);           // twelve-sided die

    cout << "rolling " << cube.NumSides() << " sided die" << endl;
    cout << cube.Roll() << endl;
    cout << cube.Roll() << endl;
    cout << "rolled " << cube.NumRolls() << " times" << endl;
```

[11]The word *dice* is the plural form of the word *die,* but a class named `Die` seems somewhat macabre. Also, using `Dice` prevents professors from jokingly saying "Die Class" to their students.

```
    cout << "rolling " << dodeca.NumSides() << " sided die" << endl;
    cout << dodeca.Roll() << endl;
    cout << dodeca.Roll() << endl;
    cout << dodeca.Roll() << endl;
    cout << "rolled " << dodeca.NumRolls() << " times" << endl;
    return 0;
}
```

roll.cpp

OUTPUT

```
prompt> roll
rolling 6 sided die
5
3
rolled 2 times
rolling 12 sided die
8
1
12
rolled 3 times

prompt> roll
rolling 6 sided die
1
6
rolled 2 times
rolling 12 sided die
8
9
2
rolled 3 times
```

`Dice` Construction. When you define a `Dice` object like `cube` or `dodeca` you must specify the number of sides for the simulated Dice object. Unlike the class `Date` which has a default (parameterless) constructor, the `Dice` class does not; you must supply the number of sides. Many people think it makes sense to have a default constructor yield a six-sided `Dice` object, so that `Dice x1,x2,x3;` defines three six-sided dice. However, when I designed the `Dice` class I decided to require a parameter. You can, of course, change the implementation of the class to permit a default constructor. We'll study how classes are implemented in the next chapter.

In C++ a constructor is a member function with the same name as the class. Constructors are functions with no return type. Neither `void`, `int`, `double`, nor any other type can be specified as the return type of a constructor. If a `Dice` variable is defined without providing arguments to the constructor as shown in *tryroll.cpp*, Program 5.12, an error message will be generated. Different compilers issue different error messages and the messages are not always intuitive for beginning programmers. However, the compilers always identify the line on which an error occurs.

Program 5.12 tryroll.cpp

```
#include <iostream>
using namespace std;
#include "dice.h"

int main()
{
    Dice spinner;

    cout << "# of sides = " << spinner.NumSides() << endl;
    return 0;
}
```

tryroll.cpp

The error message generated by the g++ compiler follows:

```
tryroll: In function ‘int main()’:
tryroll:7: no matching function for call to ‘Dice::Dice ()’
tryroll:7: in base initialization for class ‘Dice’
```

Note that the error messages indicate that the compiler tries to find a constructor with no parameters, `Dice::Dice()` but cannot find one. We'll discuss the `::` operator later. Using Metrowerks Codewarrior the error is less helpful:

```
Error   : function call ’?0()’ does not match
’Dice::Dice(int)’
’Dice::Dice(const Dice &)’
tryroll.cpp line 7   Dice spinner;
```

Using Visual C++ the error indicates that no **default constructor** can be found:

```
Compiling...
tryroll.cpp
C:\tryroll.cpp(7) : error C2512: ’Dice’ : no appropriate
                    default constructor available
Error executing cl.exe.
```

A default constructor is one with no parameters, see the error message from the g++ compiler.

Program Tip 5.6: When compilation errors occur at the point an object is constructed in a program, look carefully at the constructors in the corresponding header file to see why the error occurs. You must try to find a constructor whose parameters correspond to the the arguments passed when the object is defined.

5.4.3 Testing the Dice Class

When a new class is designed and implemented, it must be tested. Testing usually requires programs specifically designed for testing rather than for general use or for a specific application. For the `Dice` class we'd like to know whether the simulated dice behave as we'd expect real dice to behave. Are the simulated dice truly random? Do the simulated dice conform to the mathematical models that exist for random events such as dice rolls? To test the `Dice` class, we'll use a program to see whether the theoretical outcomes of rolling dice are matched by the empirical results of the test program.

We'll use a program *dicetest.cpp*, designed to toss two six-sided dice and to determine how many rolls are needed to obtain a specific sum. For example, we should expect that fewer rolls of a pair of dice are required to obtain a sum of 7 than to obtain a sum of 2. Furthermore, given that there is exactly one way to obtain a sum of 2 and one way to obtain a sum of 12 (rolling two ones and two sixes, respectively), we should expect the same number of simulated rolls to obtain either the sum of 2 or 12. The program will simulate tossing two dice and record the number of rolls needed to obtain some target between 2 and 12. We'll repeat this experiment several times and output the average number of rolls needed to obtain each sum. We wouldn't be surprised, for example, if a program needed only one roll to obtain a sum of twelve—tossing double sixes does happen. We should be surprised, however, if the experiment of trying for a twelve was repeated 1,000 times and the average number of rolls before rolling a twelve was reported to be 1—this doesn't match either our intuitive expectation or the mathematical expectation of how many rolls it takes to obtain a twelve with two six-sided dice.

Program 5.13 testdice.cpp

```
#include <iostream>
using namespace std;
#include "prompt.h"
#include "dice.h"

// simulate rolling two dice to obtain all possible sums
// repeat the "experiment" specified number of times
// Owen Astrachan, 8/9/94, modified 6/9/95, 4/20/99

double RollTest(int target,int experiments);

int main()
{
    int numTimes;                        // for one trial
    long totalRolls;                     // accumulate for all trials
    int k;

    numTimes = PromptRange("number of 'trials' ",100,20000);

    totalRolls = 0;
    for(k=2; k <= 12; k++)
    {   cout << k << "\t" << RollTest(k,numTimes) << endl;
    }

    return 0;
}

double RollTest(int target, int trials)
// precondition: 2 <= target <= 12, 0 < trials
// postcondition: returns average # of rolls needed to obtain target
//                trying 'trials' times
{
    Dice d1(6);
    Dice d2(6);

    int total = 0;
    int k;
    for(k=0; k < trials; k++)
    {   int numRolls = 1;                        //first time through loop is 1 roll
        while (d1.Roll() + d2.Roll() != target)
        {   numRolls += 1;
        }
        total += numRolls;
    }
    return double(total)/trials;
}
```

testdice.cpp

OUTPUT

```
number of 'trials'  between 100 and 20000: 10000
2       35.9015
3       18.0322
4       11.9391
5       9.0508
6       7.1973
7       5.9474
8       7.2554
9       8.9598
10      12.0036
11      17.9579
12      36.9615
```

The results obtained for trying to roll a two and a twelve are very close. Consulting a book on discrete mathematics provides an answer that is correct theoretically[12] and might further validate these empirical results. The average returned by the function `RollTest()` in Program 5.13 is converted to a `double` value by casting:

```
return double(total)/trials;
```

Casting is needed because both `total` and `trials` are `int` values and the result of dividing an `int` by an `int` value is an `int`. A `long` is used for `totalRolls` in `main` instead of an `int` because the total number of rolls over many trials will exceed the largest `int` value on 16-bit computers.

Pause to Reflect

5.39 Modify the loop in *testdice.cpp*, Program 5.13, so that the values of the dice rolls are printed for each simulated roll (run the program for only one trial). You'll need to define two integer variables to store the values of the dice rolls to print them (this can be tricky).

5.40 Write a function that rolls two `N`-sided dice and returns how many rolls are needed before the dice show the same number—that is, until doubles are rolled. The function should have one parameter: the number of sides on the dice.

5.41 Write a function that "flips a coin" (a two-sided `Dice` object) `N` times, where `N` is a parameter, and returns the number of times "heads" is flipped.

5.42 Write a function that rolls three six-sided dice and returns the number of rolls needed before all three dice show the same number. De Morgan's law may be useful in developing a loop test.

5.43 Write code that picks a random month of the year, and a random day in that month, then prints the date. The `Dice` objects you use should never cause an error. This means that for February you'll need either a 28-sided die or a 29-sided die depending on whether it's a leap year.

5.44 Write a loop to count how many times three six-sided dice must be rolled until the values showing are all different. De Morgan's law may be useful in developing a loop test.

[12] Mathematically, the expected number of rolls to obtain either a two or a twelve is 36. This is a property of independent, discrete random variables. The expected number of rolls to obtain a seven is 6.

Grace Murray Hopper *(1906–1992)*

Grace Hopper was one of the first programmers of the Harvard Mark I, the first programmable computer built in the United States. In her words she was "the third programmer on the world's first large-scale digital computer" [G9̈5]. This work was

done while she was in the Navy in the last years of World War II. It was while working on the Mark II that Hopper was involved with the first documented "bug": the famous moth inside one of the computer's relays that led to the use of the term *debugging*.

She developed the first compiler, called A-0, while working for Remington Rand in 1952. Until that time, many people believed that computers were only good for "number crunching," that computers were not capable of programming—which is what a compiler does: it produces a working program from a higher-level language. After a period of retirement, Hopper returned to naval duty in 1967, at the age of 60. She remained on active duty for 19 more years and was promoted to commodore in 1983 and to admiral in 1985. She was a proponent of innovative thinking and kept a clock on her desk that ran counterclockwise to show that things could be done differently. Although very proud of her career in the Navy, Hopper had little tolerance for bureaucracies, saying:

> *"It's better to show that something can be done and apologize for not asking permission, than to try to persuade the powers that be at the beginning."*

The Grace M. Hopper award for contributions to the field of computer science is given each year by the ACM (Association for Computing Machinery) for work done before the age of 30. In 1994 this award was given to Bjarne Stroustrup for his work in inventing and developing the language C++.

For more information see [Sla87], from which some of this biography is taken.

5.5 Chapter Review

In this chapter we discussed how classes are implemented. We also covered different looping and selection statements. Guidelines were given to assist in determining what kind of loop statement should be used and how loops are developed. The important topics covered in this chapter are summarized here.

- Interface (`.h` file) and implementation (`.cpp` files) provide an abstraction mechanism for writing and using C++ classes.
- Constructors are member functions that are automatically called to construct and initialize an object.
- Member functions are used to access an object's behavior or to get information about the object's state.
- The `for` loop is an alternative looping construct used for definite loops (where the number of iterations is known before the loop executes for the first time).
- The `do-while` loop body is always executed once, in contrast to a `while` loop body, which may never be executed.
- Infinite loops formed using `while(true)` or `for(;;)` are often used with `break` statements to avoid duplicated code and complex loop tests. However, you should be judicious in using `break` statements, because overreliance on them can lead to code that is hard to understand logically.
- A loop invariant is a statement that helps reason about and develop loops. A loop invariant is true each time the loop test is evaluated, although its truth must often be reestablished during the loop's execution.
- The built-in types `int` and `double` represent a limited range of values in computing, compared to the infinite range of values of integers and real numbers in mathematics. You must be careful to take this limited range of values into account when interpreting data and developing programs.
- Often small differences in a program can have a drastic effect on program efficiency. Determining whether a number is prime illustrates some considerations in making a program efficient.
- A `return` statement causes a function to stop, and control is returned to the calling statement. It is possible and often convenient to use `return` to exit a function early, much as a `break` statement is used to exit infinite loops.
- Fence post problems are typical in code that loops. A fence post problem is often solved using a special case before the loop or after the loop.
- The postincrement and postdecrement operators `++` and `--` are convenient shortcuts for adding and subtracting one, respectively.
- Variables modified with `const` have values that do not change. Using such constants can make programs more readable; for example, the constant `AVOGADRO` or `MOLE` carries more meaning than `6.023e23`.
- A variable is accessible only within its scope, usually delimited by curly braces: { and }. Private data variables in a class are global to all member functions of the

class.

- Constructors are special member functions used to initialize an object. A default constructor is one with no parameters. A class can have more than one constructor, like the `Date` class or only one constructor, like the `Dice` class.
- Develop test programs when you design and implement classes. Testing should be an integral part of the process of program and class design.

5.6 Exercises

5.1 Write a program modeled after the *100 bottles of X on the wall* song (see the Exercises in Chapter 3.) that will print as many verses of the song as the user specifies (both the kind of beverage and the number of bottles should be specified by the user). Try to make the program grammatical so that it doesn't print

```
one bottles of sarsaparilla on the wall
```

note the incorrect plural of bottle).

5.2 Write a program that prints a totem pole of random heads. Prompt the user for the number of heads; each head of the totem pole should be randomly drawn by using a `Dice` variable to choose among different choices for hair, eyes, mouth, etc.

O U T P U T

```
prompt> totem
how many head: 2
     |||||||/////////
     |              |
     |              |
     |    O    O    |
     |              |
    _|              |_
   |_                _|
     |   --------   |
     |              |
     ||||||||||||||||
     |              |
     |              |
     |    .    .    |
     |              |
    _|              |_
   |_                _|
     |   |______|   |
     |              |
```

5.3 Modify *testdice.cpp*, Program 5.13, so that it calculates the average number of rolls to obtain all possible sums for two *n*-sided dice, where *n* is a value entered by the user. The number of “trials should also be entered by the user. Write functions that can be used to minimize the amount of code that appears in `main`. As an example, you might consider a function with the following prototype:

```
double AverageRolls(int target, int trials, int numSides)
// pre:  2 <= target <= 2*numSides
// post: returns average # of rolls needed to obtain
//       sum 'target' rolling two dice with 'numSides'
//       sides, repeating the experiment 'trials' times
```

Can you modify this program easily to work for three *n*-sided dice rather than two?

5.4 Write a program that finds the greatest common divisor, or gcd, of two numbers. The gcd of two numbers *x* and *y* is the largest number that evenly divides both *x* and *y*. For example, the gcd of 12 and 42 is 6, and the gcd of 14 and 74 is 2. Euclid developed an algorithm for determining the gcd more than 2,000 years ago. You should use this algorithm in calculating the greatest common divisor of *x* and *y*:

```
assign r the value x % y
if r equals 0
then
   STOP, gcd is y
else
   assign x the value y
   assign y the value r
   repeat (back to top)
```

Write a function that returns the gcd of two numbers, and use the function to create a table of gcds similar to the following table, where the range of *x* and *y* are entered by the user.

```
     y
 x | 1  2  3  4  5  6  7
---+----------------------
11 | 1  1  1  1  1  1  1
12 | 1  2  3  4  1  6  1
13 | 1  1  1  1  1  1  1
14 | 1  2  1  2  1  2  7
15 | 1  1  3  1  5  3  1
```

5.5 Write a program to simulate tossing a coin (use a two-sided die). The program should toss a coin 10,000 times (or some number of times specified by the user) and keep track of the longest run of heads or tails that occurs in a sequence of simulated coin flips. Thus, in the sequence `HTHTTTHHHHT` there is a sequence of 3 tails and a sequence of 4 heads.

To keep track of the runs, four variables—`headRun`, `tailRun`, `maxHeads`, and `maxTails`—are defined and initialized to 0. These variables keep track of the length of the current head run, the length of the current tail run, and the maximum runs of heads and of tails, respectively. After the statement `heads++`, the value of `headRun` is incremented. After the statement `tails++` the value of `tailRun` is incremented. In addition, these variables must be reset to zero at the appropriate time and the values of the max head run and max tail run variables set appropriately.

5.6 Write a program that computes all **twin primes** between two values entered by the user. Twin primes are numbers that differ by two and are both primes, such as 1019 and 1021.

5.7 Write a function with prototype `int NumDigits(num)` that determines the number of digits in its parameter. Use the ideas of the previous exercise, but be sure that the function works for *all* integer values (including zero, which has one digit, and negative numbers—don't forget about the function `fabs`).

5.8 Write a boolean-valued predicate function similar to `IsPrime` that returns true if its parameter is a perfect number and false otherwise. A number is perfect if it is equal to the sum of its proper divisors (i.e., not including itself). For example, $6 = 1+2+3$ and $28 = 1+2+4+7+14$ are the first two perfect numbers. Recall that the expression `num % divisor` has value 0 exactly when `divisor` divides `num` exactly; for example, `30 % 6 == 0` but `30 % 7 = 2`. The function should be named `IsPerfect`.

5.9 Write a function `SumOfNums` that calculates and returns the sum of the numbers from 1 to *n* (where *n* is a parameter). The statement

```
cout << SumOfNums(100) << endl;
```

should cause 5050 to be printed since $1+2+\cdots+100 = 5050$. It's possible to write this program without using a loop (such a solution is often attributed to the mathematician C. F. Gauss, who supposedly discovered it when he was a boy).

5.10 The following loop sums all numbers entered by the user (and stops when the user enters a nonpositive number).

```
int num;
cin >> num;
int sum = 0;
while (num >= 0)
{   sum += num;
    cin >> num;
}
```

Explain how the two uses of `cin >>` correspond to a kind of fence post problem. Then write a program based on the foregoing loop to calculate the average of a sequence of nonnegative numbers entered by the user.

5.11 Write a function that simulates a slot machine by printing three randomly chosen strings as the values displayed by the slot machine. Each string should be chosen randomly from among four different choices, such as `"orange"`, `"lemon"`, `"lime"`, `"cherry"` (but any words will do). Choose the random values eight times and display each choice of three as shown in the following sample run. If the strings are all the same or are all different when the final sequence of these strings appears, then print a message that the user wins; otherwise the user loses.

O U T P U T

```
prompt> slots
Welcome to the slot machine simulation
Here's a spin....
cherry orange cherry
lime   lemon  cherry
lime   lemon  lemon
lime   cherry cherry
lemon  lime   cherry
lemon  lemon  lime
orange lime   lime
you lose!!

prompt> slots
Welcome to the slot machine simulation
Here's a spin....
lime   lime   orange
orange cherry orange
orange cherry lime
cherry orange lime
lime   orange orange
cherry orange lemon
lemon  lemon  lemon
all values equal, you win!!

prompt> slots
Welcome to the slot machine simulation
Here's a spin....
lemon  cherry orange
lemon  orange lemon
cherry orange lime
lime   cherry lime
cherry cherry cherry
orange cherry cherry
orange lime   cherry
all values different, you win!!
```

5.12 Using the class `BigInt` make a table of how many times each of the digits $0 \ldots 9$ occurs in huge numbers like 200! or 2^{5000}. You can determine digits by peeling off digits one at a time, as in *digits.cpp*, Program 5.5, or you can use the `BigInt` member function `ToString()` which returns a string of digits, such as `"1234567"` for the value 1,234,567, then look at each character of the string.

5.13 Write a program that displays the prime factors of a number. The prime factors of 60 are $2 \times 2 \times 3 \times 5$. Use the program to display the prime factors of all numbers between two user-entered numbers.

June 7, 1999 10:10 owltex Sheet number 71 Page number 207 magenta black

5.14 Consider the following U.S. holidays:

- Mother's Day, the second Sunday in May
- Labor Day, the first Monday in September
- Thanksgiving, the fourth Thursday in November

Write one function that determines the date on which these holidays fall in any year. The same function should be called with different parameters for the different holidays. For example, for Labor Day you would pass parameters `"Monday"`, 1, and 9 for the first Monday in September (the ninth month); for Mother's day you would pass `"Sunday"`, 2, and 5 (for May).
Use this function and write code to determine how many school days (Mon–Fri) there are between Labor Day and Thanksgiving in any year.

5.15 Daylight-saving time causes clocks to be reset in the spring and fall in many (but not all) parts of the United States. Daylight saving begins on the first Sunday of April (set clocks ahead one hour, "spring ahead") and ends on the last Sunday of October (set clocks back one hour, "fall back"). Write a program that shows the number of days in which daylight-saving time is in effect for all years from 1990 to 2010. You may find it useful to write a function that returns the number of daylight-saving days given the year (as a parameter).

5.16 Some people believe that our physical, emotional, and intellectual habits are governed by *biorhythms*. A biorhythm cycle exists for each of these three traits; the length of the cycle differs, but all cycles start when we are born. The physical cycle is 23 days long, the intellectual cycle is 33 days long, and the emotional cycle is 28 days long. The cycles repeat as sine waves, with the period of each wave given by the cycle length. A critical day occurs when all three cycles cross at the equivalent of $y = 0$ if the cycles are plotted on x and y axes. When a cycle is at its peak (e.g., as $\sin(\pi/2)$ is the peak of a sine wave), we are favored for that cycle, so that a peak on the intellectual cycle is a good day to take an exam.
Use the `Date` class to determine when your next critical day is and when your next peak and low days are for each of the three cycles.

5.17 Here are rules for one version of the game of craps, played with six-sided dice.

> A player rolls two dice. If the sum of the two is 7 or 11, the roller wins immediately; if the sum is a 2, 3, or 12, the roller loses at once. If the sum is 4, 5, 6, 8, 9, or 10, the roller rolls again. By repeating the initial number, the roller "makes his or her point" and wins. By rolling a 7 the roller "craps out" and loses. Otherwise, the roller keeps on rolling again until he or she wins or loses.

Write a program that simulates a game of craps, then modify the program to simulate 10,000 games, reporting how many simulated games are "won."

5.18 Write a program that prints a calendar for any month in any year as shown below.

OUTPUT

```
prompt> calendar
enter month between 1 and 12: 6
enter year between 1752 and 2500: 1999

   June 1999
 Su Mo Tu We Th Fr Sa
        1  2  3  4  5
  6  7  8  9 10 11 12
 13 14 15 16 17 18 19
 20 21 22 23 24 25 26
 27 28 29 30
```

For a real challenge, make it possible for the user to specify how large the calendar should be, something like this:

```
  Su  Mo  Tu  We  Th  Fr  Sa
+---+---+---+---+---+---+---+
|   | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
| 7 | 8 | 9 | 10| 11| 12| 13|
+---+---+---+---+---+---+---+
| 14| 15| 16| 17| 18| 19| 20|
+---+---+---+---+---+---+---+
| 21| 22| 23| 24| 25| 26| 27|
+---+---+---+---+---+---+---+
| 28| 29| 30|   |   |   |   |
+---+---+---+---+---+---+---+
```

or like this

```
  Sunday    Monday    Tuesday
+---------+---------+---------+
|       1 |       2 |       3 |
|         |         |         |   ...
|         |         |         |
|         |         |         |
+---------+---------+---------+
```

5.19 Write a program to track the number of times each sum for two 12-sided dice occurs over 10,000 rolls, or more generally, the number of times each sum for two N-sided dice occurs. We'll learn how to do this simply in Chapter 8, but with the programming tools you have, you'll need to write a program to write the program for you! Write a program, named *metadice.cpp*, that reads the number of sides of the dice and outputs *a program* that can be compiled and executed. For example, the following function might be part of the program; it defines and initializes variables to track each dice sum:

```
void Definitions(int sides)
// post: variable definitions for c2, c3, ...
//       are output int cX = 0;   2 <= x <= 2*sides
{
    int k;
    for(k=2; k <= 2*sides; k++)
    {   cout << "\t" << "int c" << k << " = 0;" << endl;
    }
    cout << endl << endl;
}
```

The function `Definitions` creates the variable definitions shown below; these are part of the program *that is output* by the program *metadice.cpp* that you write (and of which the function `Definitions` is a part).

O U T P U T

```
prompt> metadice
enter # sides: 5
#include <iostream>
using namespace std;
#include "dice.h"

int main()
{
        int c2 = 0;
        int c3 = 0;
        int c4 = 0;
        int c5 = 0;
        int c6 = 0;
        int c7 = 0;
        int c8 = 0;
        int c9 = 0;
        int c10 = 0;

        program continues here

        return 0;
}
```