The `while` Statement

The **`while`** statement is the simplest of C++'s iterative control statements and has the following form:

```
while ( condition ) {
      statements to be repeated
}
```

When C++ encounters a **`while`** statement, it begins by evaluating the condition in parentheses.

If the value of *condition* is **`true`**, C++ executes the statements in the body of the loop.

At the end of each cycle, C++ reevaluates *condition* to see whether its value has changed. If *condition* evaluates to **`false`**, C++ exits from the loop and continues with the statement following the closing brace at the end of the **`while`** body.

The `for` Statement

The **`for`** statement in C++ is a particularly powerful tool for specifying the control structure of a loop independently from the operations the loop body performs. The syntax looks like this:

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

C++ evaluates a **`for`** statement by executing the following steps:

1. Evaluate *init*, which typically declares a ***control variable***.
2. Evaluate *test* and exit from the loop if the value is **`false`**.
3. Execute the statements in the body of the loop.
4. Evaluate *step*, which usually updates the control variable.
5. Return to step 2 to begin the next loop cycle.

Comparing `for` and `while`

The **`for`** statement

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

is functionally equivalent to the following code using **`while`**:

```
init;
while ( test ) {
    statements to be repeated
    step;
}
```

The advantage of the **`for`** statement is that everything you need to know to understand how many times the loop will run is explicitly included in the header line.

Exercise: Reading `for` Statements

Describe the effect of each of the following **`for`** statements:

1.
```
for (int i = 1; i <= 10; i++)
```

This statement executes the loop body ten times, with the control variable **`i`** *taking on each successive value between 1 and 10.*

2.
```
for (int i = 0; i < N; i++)
```

This statement executes the loop body **`N`** *times, with* **`i`** *counting from* **`0`** *to* **`N-1`**. *This version is the standard Repeat-N-Times idiom.*

3.
```
for (int n = 99; n >= 1; n -= 2)
```

This statement counts backward from 99 to 1 by twos.

4.
```
for (int x = 1; x <= 1024; x *= 2)
```

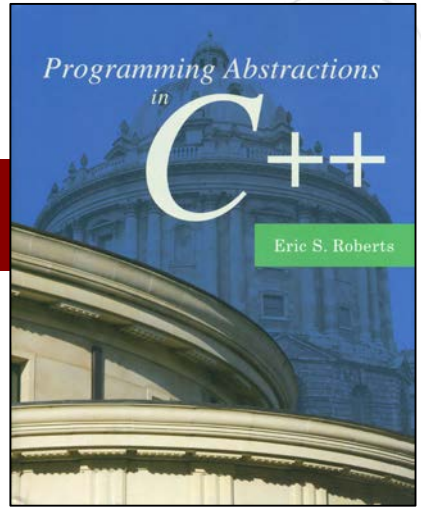
This statement executes the loop body with the variable **`x`** *taking on successive powers of two from 1 up to 1024.*

CHAPTER 2

Functions and Libraries

Your library is your paradise.

—Desiderius Erasmus, *Fisher's Study at Rotterdam,* 1524



THE UNIVERSITY OF ALABAMA
DEPARTMENT OF COMPUTER SCIENCE

The Idea of a Function

- Functions are a familiar concept from algebra, where they specify a mathematical calculation based on one or more unknown values, which are called *arguments*.
- As an example, the function

$$f(x) = x^2 + 1$$

  specifies that you can calculate the value of the function f by squaring the argument x and then adding 1 to the result. Thus, $f(0)$ is 1, $f(1)$ is 2, $f(2)$ is 5, and so on.

- In C++, you can implement a mathematical function by encoding the calculation inside a function definition, like this:

```
double f(double x) {
   return x * x + 1;
}
```

The Advantages of Using Functions

- Functions allow you to shorten a program by allowing you to include the code for a particular operation once and then use it as often as you want in a variety of different contexts, typically with different arguments.
- Functions make programs easier to read by allowing you to invoke an entire sequence of operations using a single name that corresponds to a higher-level understanding of the purpose of the function.
- Functions simplify program maintenance by allowing you to divide a large program into smaller, more manageable pieces. This process is called *decomposition.*

Functions and Algorithms

- Functions are critical to programming because they provide a structure in which to express algorithms.
- Algorithms for solving a particular problem can vary widely in their efficiency. It makes sense to think carefully when you are choosing an algorithm because making a bad choice can be extremely costly.
- The next few slides illustrate this principle by implementing two algorithms for computing the *greatest common divisor* of the integers *x* and *y,* which is defined to be the largest integer that divides evenly into both.

The Brute-Force Approach

- One strategy for computing the greatest common divisor is to count backwards from the smaller value until you find one that divides evenly into both. The code looks like this:

```
int gcd(int x, int y) {
   int guess = x;
   while (x % guess != 0 || y % guess != 0) {
      guess--;
   }
   return guess;
}
```

- This algorithm must terminate for positive values of `x` and `y` because the value of `guess` will eventually reach 1. At that point, `guess` must be the greatest common divisor because the `while` loop will have already tested all larger ones.
- Trying every possibility is called a *brute-force strategy.*

Euclid's Algorithm

touching lives

- If you use the brute-force approach to compute the greatest common divisor of 1000005 and 1000000, the program will take a million steps to tell you the answer is 5.
- You can get the answer much more quickly if you use a better algorithm. The Greek mathematician Euclid of Alexandria described a more efficient algorithm 23 centuries ago, which looks like this:

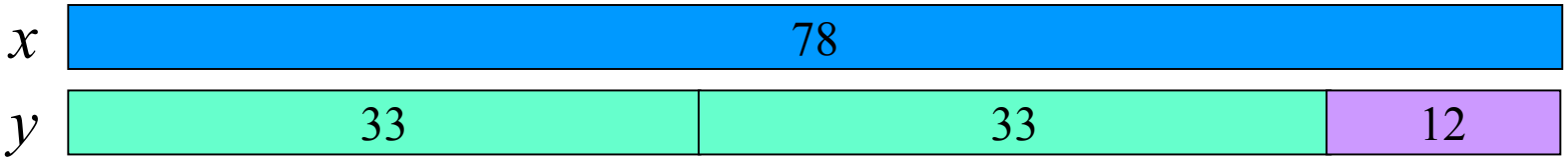
```
int gcd(int x, int y) {
   int r = x % y;
   while (r != 0) {
      x = y;
      y = r;
      r = x % y;
   }
   return y;
}
```

How Euclid's Algorithm Works

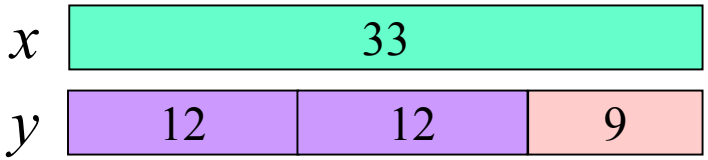
- If you use Euclid's algorithm on 1000005 and 1000000, you get the correct answer in just two steps, which is much better than the million steps required by brute force.
- Euclid's great insight was that the greatest common divisor of x and y must also be the greatest common divisor of y and the remainder of x divided by y. He was, moreover, able to prove this proposition in Book VII of his *Elements*.
- It is easy to see how Euclid's algorithm works if you think about the problem geometrically, as Euclid did. The next slide works through the steps in the calculation when x is 78 and y is 33.

An Illustration of Euclid's Algorithm

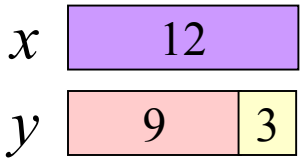
Step 1: Compute the remainder of 78 divided by 33:



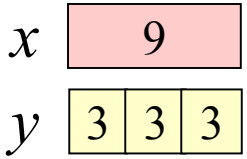
Step 2: Compute the remainder of 33 divided by 12:



Step 3: Compute the remainder of 12 divided by 9:



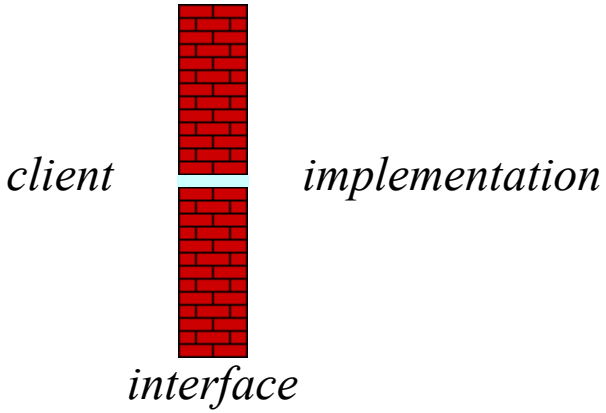
Step 4: Compute the remainder of 9 divided by 3:



Because there is no remainder, the answer is 3.

Libraries

- Modern programming depends on the use of libraries. When you create a typical application, you write only a tiny fraction of the code.
- Libraries can be viewed from two perspectives. Code that uses a library is called a *client*. The code for the library itself is called the *implementation*.
- The point at which the client and the implementation meet is called the *interface,* which serves as both a barrier and a communication channel:



Useful Functions in the <cmath> Library

`abs(`*x*`)`	Returns the absolute value of *x*.
`sqrt(`*x*`)`	Returns the square root of *x*.
`floor(`*x*`)`	Returns the largest integer less than or equal to *x*.
`ceil(`*x*`)`	Returns the smallest integer greater than or equal to *x*.

`exp(`*x*`)`	Returns the exponential function of *x* (e^x).
`log(`*x*`)`	Returns the natural logarithm (base e) of *x*.
`log10(`*x*`)`	Returns the common logarithm (base 10) of *x*.
`pow(`*x*`, `*y*`)`	Returns x^y.

`cos(`*theta*`)`	Returns the trigonometric cosine of the radian angle *theta*.
`sin(`*theta*`)`	Returns the sine of the radian angle *theta*.
`tan(`*theta*`)`	Returns the tangent of the radian angle *theta*.
`atan(`*x*`)`	Returns the principal arctangent of *x*.
`atan2(`*y*`, `*x*`)`	Returns the arctangent of *y* divided by *x*.

Defining Functions in C++

- The general form of a function definition in C++ looks much the same as it does in other languages derived from C, such as Java:

```
type  name (parameter list)  {
      statements in the function body
}
```

where *type* indicates what type the function returns, *name* is the name of the function, and *parameter list* is a list of variable declarations used to hold the values of each argument.

- All functions need to be declared before they are called by specifying a *prototype* consisting of the header line followed by a semicolon.

C++ Enhancements to Functions

- Functions can be *overloaded,* which means that you can define several different functions with the same name as long as the correct version can be determined by looking at the number and types of the arguments. The pattern of arguments required for a particular function is called its *signature*.
- Functions can specify *optional parameters* by including an initializer after the variable name. For example, the function prototype

```
void setMargin(int margin = 72);
```

Indicates that `setMargin` takes an optional argument that defaults to 72.

Computing Factorials

touching lives

- The *factorial* of a number *n* (which is usually written as *n*! in mathematics) is defined to be the product of the integers from 1 up to *n*. Thus, 5! is equal to 120, which is 1×2×3×4×5.
- The following function definition uses a `for` loop to compute the factorial function:

```
int fact(int n) {
   int result = 1;
   for (int i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}
```

The Mechanics of Calling a Function

When you invoke a function, the following actions occur:

1. C++ evaluates the argument expressions in the context of the calling function.
2. C++ then copies each argument value into the corresponding parameter variable, which is allocated in a newly assigned region of memory called a *stack frame*. This assignment follows the order in which the arguments appear: the first argument is copied into the first parameter variable, and so on.
3. C++ then evaluates the statements in the function body, using the new stack frame to look up the values of local variables.
4. When C++ encounters a `return` statement, it computes the return value and substitutes that value in place of the call.
5. C++ then discards the stack frame for the called function and returns to the caller, continuing from where it left off.

The Combinations Function

touching lives

- To illustrate function calls, the text uses a function $C(n, k)$ that computes the *combinations* function, which is the number of ways one can select k elements from a set of n objects.
- Suppose, for example, that you have a set of five coins: a penny, a nickel, a dime, a quarter, and a dollar:

How many ways are there to select two coins?

penny + nickel	nickel + dime	dime + quarter	quarter + dollar
penny + dime	nickel + quarter	dime + dollar	
penny + quarter	nickel + dollar		
penny + dollar			

for a total of 10 ways.

Combinations and Factorials

- Fortunately, mathematics provides an easier way to compute the combinations function than by counting all the ways. The value of the combinations function is given by the formula

$$C(n,k) = \frac{n!}{k! \times (n-k)!}$$

- Given that you already have a `fact` function, is easy to turn this formula directly into a C++ function, as follows:

```
int combinations(int n, int k) {
   return fact(n) / (fact(k) * fact(n - k));
}
```

- The next slide simulates the operation of `combinations` and `fact` in the context of a simple `main` function.

The Combinations Program


```
int main() {
   int n, k;
   cout << "Enter the number of objects (n): ";
   cin >> n;
   cout << "Enter the number to be chosen (k): ";
   cin >> k;
   cout << "C(n, k) = " << combinations(n, k) << endl;
   return 0;
}
```

Combinations

```
Enter number of objects in the set (n): 5
Enter number to be chosen (k): 2
C(5, 2) = 10
```

Reference Parameters

- C++ allows callers and functions to share information using a technique known as *call by reference.*
- C++ indicates call by reference by adding an ampersand (`&`) before the parameter name. A single function often has both *value parameters* and *reference parameters,* as illustrated by the `solveQuadratic` function:

```
void solveQuadratic(double a, double b, double c,
                    double & x1, double & x2);
```

- Call by reference has two primary purposes:
  - It creates a sharing relationship that makes it possible to pass information in both directions through the parameter list.
  - It increases efficiency by eliminating the need to copy an argument. This consideration becomes more important when the argument is a large object.

Call by Reference Example

- The following function swaps the values of two integers:

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

- The arguments to `swap` must be assignable objects, which for the moment means variables.
- If you left out the `&` characters in the parameter declarations, calling this function would have no effect on the calling arguments because the function would exchange local copies.

The End