Module 5
Introducing Functions

Table of Contents

This module begins an in-depth discussion of the function. Functions are the building blocks of C++, and a firm understanding of them is fundamental to becoming a successful C++ programmer. Here, you will learn how to create a function. You will also learn about passing arguments, returning values, local and global variables, function prototypes, and recursion.

Function Fundamentals

A function is a subroutine that contains one or more C++ statements and performs a specific task. Every program that you have written so far has used one function: main( ). They are called the building blocks of C++ because a program is a collection of functions. All of the "action" statements of a program are found within functions. Thus, a function contains the statements that you typically think of as being the executable part of a program. Although very simple programs, such as many of those shown in this book, will have only a main( ) function, most programs will contain several functions. In fact, a large, commercial program will define hundreds of functions.

C++ A Beginner's Guide by Herbert Schildt

CRITICAL SKILL 5.1: Know the general form of a function

All C++ functions share a common form, which is shown here:

return-type name(parameter-list) { // body of function }

Here, return-type specifies the type of data returned by the function. This can be any valid type, except an array. If the function does not return a value, its return type must be void. The name of the function is specified by name. This can be any legal identifier that is not already in use. The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the function when it is called. If the function has no parameters, then the parameter list will be empty.

Braces surround the body of the function. The function body is composed of the C++ statements that define what the function does. The function terminates and returns to the calling code when the closing curly brace is reached.

CRITICAL SKILL 5.2: Creating a Function

It is easy to create a function. Since all functions share the same general form, they are all similar in structure to the main( ) functions that you have been using. Let's begin with a simple example that contains two functions: main( ) and myfunc( ). Before running this program (or reading the description that follows), examine it closely and try to figure out exactly what it displays on the screen.

```cpp
// This program contains two functions: main() and myfunc().

#include <iostream>
using namespace std;

void myfunc(); // myfunc's prototype          This is the prototype for myfunc( ).

int main()
{
  cout << "In main()\n";

  myfunc(); // call myfunc()

  cout << "Back in main()\n";

  return 0;
}

// This is the function's definition.
void myfunc()
{
  cout << "Inside myfunc()\n";          This is the function called myfunc( ).
}
```
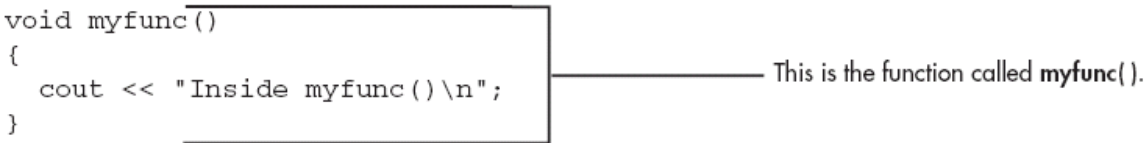
The program works like this. First, main( ) begins, and it executes the first cout statement. Next, main( ) calls myfunc( ). Notice how this is achieved: the function's name is followed by parentheses. In this case, the function call is a statement and, therefore, must end with a semicolon. Next, myfunc( ) executes its cout statement and then returns to main( ) when the closing } is encountered. In main( ), execution resumes at the line of code immediately following the call to myfunc( ). Finally, main( ) executes its second cout statement and then terminates. The output is shown here:

In main()
Inside myfunc()
Back in main()

The way myfunc( ) is called and the way that it returns represent a specific instance of a process that applies to all functions. In general, to call a function, specify its name followed by parentheses. When a function is called, execution jumps to the function. Execution continues inside the function until its closing curly brace is encountered. When the function ends, program execution returns to the caller at the statement immediately following the function call.

Notice this statement in the preceding program:

void myfunc(); // myfunc's prototype

As the comment states, this is the prototype for myfunc( ). Although we will discuss prototypes in detail later, a few words are necessary now. A function prototype declares the function prior to its definition. The prototype allows the compiler to know the function's return type, as well as the number and type of any parameters that the function may have. The compiler needs to know this information prior to the first time the function is called. This is why the prototype occurs before main( ). The only function that does not require a prototype is main( ), since it is predefined by C++.

The keyword void, which precedes both the prototype for myfunc( ) and its definition, formally states that myfunc( ) does not return a value. In C++, functions that don't return values are declared as void.

CRITICAL SKILL 5.3: Using Arguments

It is possible to pass one or more values to a function that you create. A value passed to a function is called an argument. Thus, arguments are a way to get information into a function.

When you create a function that takes one or more arguments, variables that will receive those arguments must also be declared. These variables are called the parameters of the function. Here is an example that defines a function called box( ) that computes the volume of a box and displays the result. It has three parameters.

void box(int length, int width, int height)

{ cout << "volume of box is " << length * width * height << "\n";

}

In general, each time box( ) is called, it will compute the volume by multiplying the values passed to its parameters: length, width, and height. Notice how the parameters are declared. Each parameter's declaration is separated from the next by a comma, and the parameters are contained within the parentheses that follow the function's name. This same basic approach applies to all functions that use parameters.

To call box( ), you must specify three arguments. For example:

box(7, 20, 4); box(50, 3, 2); box(8, 6, 9);

The values specified between the parentheses are arguments passed to box( ), and the value of each argument is copied into its matching parameter. Therefore, in the first call to box( ), 7 is copied into length, 20 is copied into width, and 4 is copied into height. In the second call, 50 is copied into length, 3 into width, and 2 into height. In the third call, 8 is copied into length, 6 into width, and 9 into height.

The following program demonstrates box( ):

```
// A simple program that demonstrates box().

#include <iostream>
using namespace std;

void box(int length, int width, int height); // box()'s prototype

int main()
{
  box(7, 20, 4);          Pass arguments to box( ).
  box(50, 3, 2);
  box(8, 6, 9);

  return 0;                          These parameters receive
}                                    the values of the arguments
                                     passed to box( ).
// Compute the volume of a box.
void box(int length, int width, int height)
{
  cout << "volume of box is " << length * width * height << "\n";
}
```

The output from the program is shown here:

```
volume of box is 560
volume of box is 300
volume of box is 432
```

Remember the term argument refers to the value that is used to call a function. The variable that receives the value of an argument is called a parameter. In fact, functions that take arguments are called parameterized functions.

1. When a function is called, what happens to program execution?
2. What is the difference between an argument and a parameter?
3. If a function requires a parameter, where is it declared?

CRITICAL SKILL 5.4: Using return

In the preceding examples, the function returned to its caller when its closing curly brace was encountered. While this is acceptable for many functions, it won't work for all. Often, you will want to control precisely how and when a function returns. To do this, you will use the return statement.

The return statement has two forms: one that returns a value, and one that does not. We will begin with the version of return that does not return a value. If a function has a void return type (that is, if the function does not return a value), then it can use this form of return:

return;

When return is encountered, execution returns immediately to the caller. Any code remaining in the function is ignored. For example, consider this program:

```
void f();

int main()
{
  cout << "Before call\n";

  f();

  cout << "After call\n";

  return 0;
}

// A void function that uses return.
void f()
{
  cout << "Inside f()\n";

  return; // return to caller  ←——————— This causes an immediate
                                         return, bypassing the
                                         remaining cout statement.
  cout <<"This won't display.\n";
}
```

The output from the program is shown here:

Introducing Functions

Before call
Inside f()
After call

As the output shows, f( ) returns to main( ) as soon as the return statement is encountered. The second cout statement is never executed.

Here is a more practical example of return. The power( ) function shown in the next program displays the outcome of an integer raised to a positive integer power. If the exponent is negative, the return statement causes the function to terminate before any attempt is made to compute the result.

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

void power(int base, int exp);

int main()
{
  power(10, 2);
  power(10, -2);
  return 0;
}

// Raise an integer to a positive power.
void power(int base, int exp)
{
  int i;                              Return when exp is negative.

  if(exp < 0) return; /* Can't do negative exponents. */

  i = 1;

  for( ; exp; exp--) i = base * i;
  cout << "The answer is: " << i;
}
```

The output from the program is shown here:

The answer is: 100

When exp is negative (as it is in the second call), power( ) returns, bypassing the rest of the function.

A function may contain several return statements. As soon as one is encountered, the function returns. For example, this fragment is perfectly valid:

```
void f()
{
  // ...

  switch(c) {
    case 'a': return;
    case 'b': // ...
    case 'c': return;
  }
  if(count < 100) return;
  // ...
}
```

Be aware, however, that having too many returns can destructure a function and confuse its meaning. It is best to use multiple returns only when they help clarify a function.

Returning Values

A function can return a value to its caller. Thus, a return value is a way to get information out of a function. To return a value, use the second form of the return statement, shown here:

return value;

Here, value is the value being returned. This form of the return statement can be used only with functions that do not return void.

A function that returns a value must specify the type of that value. The return type must be compatible with the type of data used in the return statement. If it isn't, a compile-time error will result. A function can be declared to return any valid C++ data type, except that a function cannot return an array.

To illustrate the process of functions returning values, the box( ) function can be rewritten as shown here. In this version, box( ) returns the volume. Notice that the placement of the function on the right side of an assignment statement assigns the return value to a variable.

```
// Returning a value.

#include <iostream>
using namespace std;

int box(int length, int width, int height); // return the volume

int main()
{                                   ┌─── The return value from box( )
  int answer;                       │    is assigned to answer.
                                    ▼
  answer = box(10, 11, 3); // assign return value
  cout << "The volume is " <<  answer;

  return 0;
}

// This function returns a value.
int box(int length, int width, int height)
{
  return length * width * height ;  ◄───── Return the volume.
}
```

Here is the output:

The volume is 330

In this example, box( ) returns the value of length * width * height using the return statement. This value is then assigned to answer. That is, the value returned by the return statement becomes box( )'s value in the calling routine.

Since box( ) now returns a value, it is not preceded by the keyword void. (Remember, void is only used when a function does not return a value.) Instead, box( ) is declared as returning a value of type int. Notice that the return type of a function precedes its name in both its prototype and its definition.

Of course, int is not the only type of data a function can return. As stated earlier, a function can return any type of data except an array. For example, the following program reworks box( ) so that it takes double parameters and returns a double value:

```cpp
// Returning a double value.

#include <iostream>
using namespace std;

// use double data
double box(double length, double width, double height);

int main()
{
  double answer;

  answer = box(10.1, 11.2, 3.3); // assign return value
  cout << "The volume is " <<  answer;

  return 0;
}

// This version of box uses double data.
double box(double length, double width, double height)
{
  return length * width * height ;
}
```

Here is the output:

The volume is 373.296

One more point: If a non-void function returns because its closing curly brace is encountered, an undefined (that is, unknown) value is returned. Because of a quirk in the formal C++ syntax, a non-void function need not actually execute a return statement. This can happen if the end of the function is reached prior to a return statement being encountered. However, because the function is declared as returning a value, a value will still be returned—even though it is just a garbage value. Of course, good practice dictates that any non-void function that you create should return a value via an explicit return statement.

CRITICAL SKILL 5.5: Using Functions in Expressions

In the preceding example, the value returned by box( ) was assigned to a variable, and then the value of this variable was displayed via a cout statement. While not incorrect, these programs could be written more efficiently by using the return value directly in the cout statement. For example, the main( ) function in the preceding program can be written more efficiently like this:

```
int main()
{
  // use the return value of box( ) directly
  cout << "The volume is " <<  box(10.1, 11.2, 3.3);

  return 0;
}
```

Use the return value of **box( )** directly in a **cout** statement.

When the cout statement executes, box( ) is automatically called so that its return value can be obtained. This value is then output. There is no reason to first assign it to some variable.

In general, a non-void function can be used in any type of expression. When the expression is evaluated, the function is automatically called so that its return value can be obtained. For example, the following program sums the volume of three boxes and then displays the average volume:

```
// Use box() in an expression.

#include <iostream>
using namespace std;

// use double data
double box(double length, double width, double height);

int main()
{
  double sum;

  sum = box(10.1, 11.2, 3.3) + box(5.5, 6.6, 7.7) +
        box(4.0, 5.0, 8.0);

  cout << "The sum of the volumes is " <<  sum << "\n";
  cout << "The average volume is " << sum / 3.0 << "\n";

  return 0;
}

// This version of box uses double data.
double box(double length, double width, double height)
{
  return length * width * height ;
}
```

The output of this program is shown here:

The sum of the volumes is 812.806 The average volume is 270.935

Progress Check

1. Show the two forms of the return statement.

2. Can a void function return a value?

3. Can a function call be part of an expression?

Scope Rules

Up to this point, we have been using variables without formally discussing where they can be declared, how long they remain in existence, and what parts of a program have access to them. These attributes are determined by the scope rules defined by C++.

In general, the scope rules of a language govern the visibility and lifetime of an object.

Although C++ defines a finely grained system of scopes, there are two basic ones: local and global. In both of these scopes, you can declare variables. In this section, you will see how variables declared in a local scope differ from variables declared in the global scope, and how each relates to the function.

CRITICAL SKILL 5.6: Local Scope

A local scope is created by a block. (Recall that a block begins with an opening curly brace and ends with a closing curly brace.) Thus, each time you start a new block, you are creating a new scope. A variable can be declared within any block. A variable that is declared inside a block is called a local variable.

A local variable can be used only by statements located within the block in which it is declared. Stated another way, local variables are not known outside their own code blocks.

Thus, statements defined outside a block cannot access an object defined within it. In essence, when you declare a local variable, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

One of the most important things to understand about local variables is that they exist only while the block of code in which they are declared is executing. A local variable is created when its declaration statement is encountered within its block, and destroyed when the block is left. Because a local variable is destroyed upon exit from its block, its value is lost. The most common code block in which variables are declared is the function. Each function defines a block of code that begins with the function's opening curly brace and ends with its closing curly brace. A function's code and data are private to that function and cannot be accessed by any statement in any other function except through a call to that function. (It is not possible, for instance, to use a goto statement to jump into the middle of another function.)

The body of a function is hidden from the rest of the program, and it can neither affect nor be affected by other parts of the program. Thus, the contents of one function are completely separate from the contents of another. Stated another way, the code and data that are defined within one function cannot interact with the code or data defined in another function, because the two functions have a different scope. Because each function defines its own scope, the variables declared within one function have no effect on those declared in another—even if those variables share the same name.

For example, consider the following program:

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

void f1();

int main()
{
  int val = 10;          // This val is local to main().

  cout << "val in main(): " << val << '\n';
  f1();
  cout << "val in main(): " << val << '\n';

  return 0;
}

void f1()
{
  int val = 88;          // This val is local to f1().

  cout << "val in f1(): " << val << "\n";
}
```
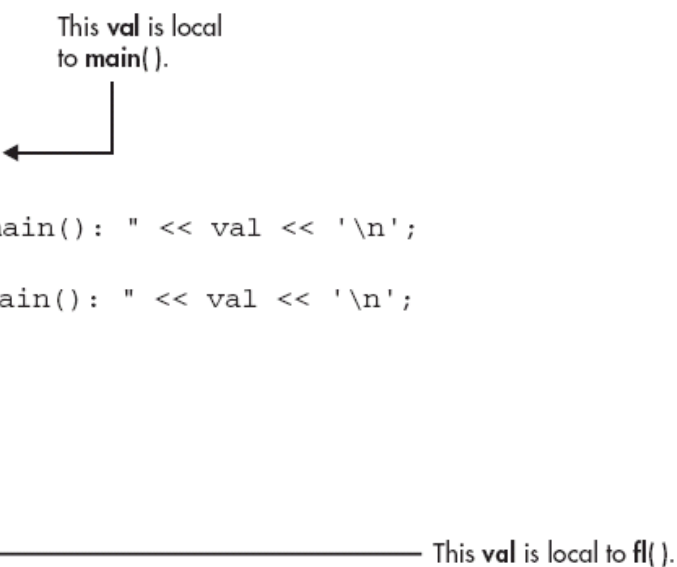
Here is the output:

val in main(): 10 val in f1(): 88 val in main(): 10

An integer called val is declared twice, once in main( ) and once in f1( ). The val in main( ) has no bearing on, or relationship to, the one in f1( ). The reason for this is that each val is known only to the function in which it is declared. As the output shows, even though the val declared in f1( ) is set to 88, the content of val in main( ) remains 10.

Because a local variable is created and destroyed with each entry and exit from the block in which it is declared, a local variable will not hold its value between activations of its block. This is especially important to remember in terms of a function call. When a function is called, its local variables are created. Upon its return, they are destroyed. This means that local variables cannot retain their values between calls.

If a local variable declaration includes an initializer, then the variable is initialized each time the block is entered. For example:

```
/*
    A local variable is initialized each
    time its block is entered.
*/

#include <iostream>
using namespace std;

void f();

int main()
{
  for(int i=0; i < 3; i++) f();

  return 0;
}

// num is initialized each time f() is called.
void f()
{
    int num = 99;    ←——————— num is set to 99 each time f( ) is called.

    cout << num << "\n";

    num++; // this has no lasting effect
}
```

The output shown here confirms that num is initialized each time f( ) is called:

99 99 99

A local variable that is not initialized will have an unknown value until it is assigned one.

Local Variables Can Be Declared Within Any Block

It is common practice to declare all variables needed within a function at the beginning of that function's code block. This is done mainly so that anyone reading the code can easily determine what variables are used. However, the beginning of the function's block is not the only place where local variables can be declared. A local variable can be declared anywhere, within any block of code. A variable declared within a block is local to that block. This means that the variable does not exist until the block is entered and is destroyed when the block is exited. Furthermore, no code outside that block—including other code in the function— can access that variable. To understand this, try the following program:

```
// Variables can be local to a block.

#include <iostream>
using namespace std;

int main() {
  int x = 19; // x is known to all code.

  if(x == 19) {
    int y = 20; ◄──────────── y is local to the if block.

    cout << "x + y is " << x + y << "\n";
  }

  //  y = 100; // Error!  y not known here.

  return 0;
}
```

The variable x is declared at the start of main( )'s scope and is accessible to all subsequent code within main( ). Within the if block, y is declared. Since a block defines a scope, y is visible only to other code within its block. This is why outside of its block, the line

y = 100;

is commented out. If you remove the leading comment symbol, a compile-time error will occur, because y is not visible outside of its block. Within the if block, x can be used because code within a block has access to variables declared by an enclosing block.

Although local variables are typically declared at the beginning of their block, they need not be. A local variable can be declared anywhere within a block as long as it is declared before it is used. For example, this is a perfectly valid program:

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

int main()
{
  cout << "Enter a number: ";
  int a; // declare one variable
  cin >> a;

  cout << "Enter a second number: ";
  int b; // declare another variable
  cin >> b;

  cout << "Product: " << a*b << '\n';

  return 0;
}
```
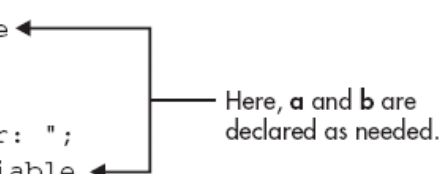
Here, **a** and **b** are declared as needed.

In this example, a and b are not declared until just before they are needed. Frankly, most programmers declare local variables at the beginning of the function that uses them, but this is a stylistic issue.

Name Hiding

When a local variable declared in an inner block has the same name as a variable declared in an outer block, the variable declared in the inner block hides the one in the outer block. For example:

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

int main()
{
  int i;
  int j;

  i = 10;
  j = 100;

  if(j > 0) {
    int i; // this i is separate from outer i

    i = j / 2;
    cout << "inner i: " << i << '\n';
  }

  cout << "outer i: " << i << '\n';

  return 0;
}
```
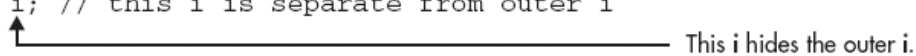
This **i** hides the outer **i**.

The output from this program is shown here:

inner i: 50

outer i: 10

The i declared within the if block hides the outer i. Changes that take place on the inner i have no effect on the outer i. Furthermore, outside of the if block, the inner i is unknown and the outer i comes back into view.

Function Parameters

The parameters to a function are within the scope of the function. Thus, they are local to the function. Except for receiving the values of the arguments, parameters behave like any other local variables.

Ask the Expert

Q: What does the keyword auto do? I have heard that it is used to declare local variables. Is this right?

A: The C++ language contains the keyword auto, which can be used to declare local variables. However, since all local variables are, by default, assumed to be auto, it is virtually never used. Thus, you will not see it used in any of the examples in this book. However, if you choose to use it, place it immediately before the variable's type, as shown here:

auto char ch;

Again, auto is optional and not used elsewhere in this book.

CRITICAL SKILL 5.7: Global Scope

Since local variables are known only within the function in which they are declared, a question may have occurred to you: How do you create a variable that can be shared by more than one function? The answer is to declare the variable in the global scope. The global scope is the declarative region that is outside of all functions. Declaring a variable in the global scope creates a global variable.

Global variables are known throughout the entire program. They can be used by any piece of code, and they maintain their values during the entire execution of the program. Therefore, their scope extends to the entire program. You can create global variables by declaring them outside of any function. Because they are global, they can be accessed by any expression, regardless of which function contains the expression.

The following program demonstrates the use of a global variable. The variable count has been declared outside of all functions. Its declaration is before the main( ) function. However, it could have been placed anywhere, as long as it was not in a function. Remember, though, that since you must declare a variable before you use it, it is best to declare global variables at the top of the program.

```
// Use a global variable.

#include <iostream>
using namespace std;

void func1();
void func2();
                          count is global.

int count; // This is a global variable.

int main()
{
  int i; // This is a local variable

  for(i=0; i<10; i++) {
    count = i * 2;   ◄————————— This refers to the global count.
    func1();
  }

  return 0;
}                              ┌—— This also refers to the global count.

void func1()
{
  cout << "count: " << count; // access global count
  cout << '\n'; // output a newline
  func2();
}

void func2()
{              ┌————————— This count is local to func2( ).
  int count; // this is a local variable

  for(count=0; count<3; count++) cout << '.';
}                              ↑————————————————— This refers to the local count.
```

The output from the program is shown here:

```
count: 0
...count: 2
...count: 4
...count: 6
...count: 8
...count: 10
...count: 12
...count: 14
...count: 16
...count: 18
...
```

Looking closely at this program, it should be clear that both main( ) and func1( ) use the global variable count. In func2( ), however, a local variable called count is declared. When func2( ) uses count, it is referring to its local variable, not the global one. It is important to understand that if a global variable and a local variable have the same name, all references to that variable name inside the function in which the local variable is declared will refer to the local variable and have no effect on the global variable. Thus, a local variable hides a global variable of the same name.

Global variables are initialized at program startup. If a global variable declaration includes an initializer, then the variable is initialized to that value. If a global variable does not include an initializer, then its value is set to zero.

Storage for global variables is in a fixed region of memory set aside for this purpose by your program. Global variables are helpful when the same data is used by several functions in your program, or when a variable must hold its value throughout the duration of the program. You should avoid using unnecessary global variables, however, for three reasons:

They take up memory the entire time your program is executing, not just when they are needed.

Using a global variable where a local variable will do makes a function less general, because it relies on something that must be defined outside itself.

Using a large number of global variables can lead to program errors because of unknown, and unwanted, side effects. A major problem in developing large programs is the accidental modification of a variable's value due to its use elsewhere in a program. This can happen in C++ if you use too many global variables in your programs.

Progress Check

1. What are the main differences between local and global variables?

2. Can a local variable be declared anywhere within a block?

3. Does a local variable hold its value between calls to the function in which it is declared?

CRITICAL SKILL 5.8: Passing Pointers and Arrays to Functions

The preceding examples have used simple values, such as int or double, as arguments. However, there will be times when you will want to use pointers and arrays as arguments. While passing these types of arguments is straightforward, some special issues need to be addressed.

Passing a Pointer

To pass a pointer as an argument, you must declare the parameter as a pointer type. Here is an example:

```
// Pass a pointer to a function.

#include <iostream>
using namespace std;                              f( ) takes an int * pointer as a parameter.

void f(int *j); // f() declares a pointer parameter

int main()
{
   int i;
   int *p;

   p = &i; // p now points to i

   f(p); // pass a pointer  ←———— f( ) is called with a pointer to an integer.

   cout << i;  // i is now 100

   return 0;
}

// f() receives a pointer to an int.
void f(int *j)
{
   *j = 100; // var pointed to by j is assigned 100
}
```

Study this program carefully. As you can see, f( ) takes one parameter: an int pointer. Inside main( ), p (an int pointer) is assigned the address of i. Next, f( ) is called with p as an argument. When the pointer parameter j receives p, it then also points to i within main( ). Thus, the assignment

*j = 100;

causes i to be given the value 100. For the general case, f( ) assigns 100 to whatever address it is called with.

In the preceding example, it is not actually necessary to use the pointer variable p. Instead, you can simply precede i with an & when f( ) is called. This causes the address of i to be passed to f( ). The revised program is shown here:

```
// Pass a pointer to a function.

#include <iostream>
using namespace std;

void f(int *j);

int main()
{
  int i;

  f(&i);  ←——————— No need for p. The address
                    of i is passed directly.

  cout << i;

  return 0;
}

void f(int *j)
{
  *j = 100; // var pointed to by j is assigned 100
}
```

It is crucial that you understand one thing about passing pointers to functions: when you perform an operation within the function that uses the pointer, you are operating on the variable that is pointed to by that pointer. Thus, the function will be able to change the value of the object pointed to by the parameter.

Passing an Array

When an array is an argument to a function, the address of the first element of the array is passed, not a copy of the entire array. (Recall that an array name without any index is a pointer to the first element in the array.) This means that the parameter declaration must be of a compatible type. There are three ways to declare a parameter that is to receive an array pointer. First, it can be declared as an array of the same type and size as that used to call the function, as shown here:

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

void display(int num[10]);

int main()
{
  int t[10], i;

  for(i=0; i < 10; ++i) t[i]=i;

  display(t); // pass array t to a function

  return 0;
}

// Print some numbers.
void display(int num[10])  ◄─────── Parameter declared
{                                   as a sized array
  int i;

  for(i=0; i < 10; i++) cout << num[i] << ' ';
}
```

Even though the parameter num is declared to be an integer array of ten elements, the C++ compiler will automatically convert it to an int pointer. This is necessary because no parameter can actually receive an entire array. Since only a pointer to the array will be passed, a pointer parameter must be there to receive it.

A second way to declare an array parameter is to specify it as an unsized array, as shown here:

```cpp
void display(int num[])  ◄─────── Parameter declared as an unsized array
{
  int i;

  for(i=0; i < 10; i++) cout << num[i] << ' ';
}
```

Here, num is declared to be an integer array of unknown size. Since C++ provides no array boundary checks, the actual size of the array is irrelevant to the parameter (but not to the program, of course). This method of declaration is also automatically transformed into an int pointer by the compiler.

The final way that num can be declared is as a pointer. This is the method most commonly used in professionally written C++ programs. Here is an example:

```
void display(int *num)
{                          Parameter declared as a pointer
   int i;

   for(i=0; i < 10; i++) cout << num[i] << ' ';
}
```

The reason it is possible to declare num as a pointer is that any pointer can be indexed

```
                              void cube(int *n, int num)
{
   while(num) {
      *n = *n * *n * *n;      This changes the value of the array
      num--;                  element pointed to by n.
      n++;
   }
}
```

using [ ], as if it were an array. Recognize that all three methods of declaring an array parameter yield the same result: a pointer.

It is important to remember that when an array is used as a function argument, its address is passed to a function. This means that the code inside the function will be operating on, and potentially altering, the actual contents of the array used to call the function. For example, in the following program examine the function cube( ), which converts the value of each element in an array into its cube. To call cube( ), pass the address of the array as the first argument and the size of the array as the second.

```
// Change the contents of an array using a function.

#include <iostream>
using namespace std;

void cube(int *n, int num);

int main()
{
   int i, nums[10];

   for(i=0; i < 10; i++) nums[i] = i+1;

   cout << "Original contents: ";
   for(i=0; i < 10; i++) cout << nums[i] << ' ';
   cout << '\n';

   cube(nums, 10); // compute cubes        Pass address of nums to cube( ).

   cout << "Altered contents: ";
   for(i=0; i<10; i++) cout << nums[i] << ' ';

   return 0;
}
```

Here is the output produced by this program:

Original contents: 1 2 3 4 5 6 7 8 9 10

Altered contents: 1 8 27 64 125 216 343 512 729 1000

As you can see, after the call to cube( ), the contents of array nums in main( ) will be cubes of its original values. That is, the values of the elements of nums have been modified by the statements within cube( ), because n points to nums.

Passing Strings

Because a string is simply a character array that is null-terminated, when you pass a string to a function, only a pointer to the beginning of the string is actually passed. This is a pointer of type char *. For example, consider the following program. It defines the function strInvertCase( ), which inverts the case of the letters within a string.

```cpp
// Pass a string to a function.

#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;

void strInvertCase(char *str);

int main()
{
  char str[80];

  strcpy(str, "This Is A Test");

  strInvertCase(str);

  cout << str; // display modified string
```

```
   return 0;
}

// Invert the case of the letters within a string.
void strInvertCase(char *str)
{
  while(*str) {

    // invert case
    if(isupper(*str)) *str = tolower(*str);
    else if(islower(*str)) *str = toupper(*str);

    str++; // move on to next char
  }
}
```

Here is the output:

tHIS iS a tEST

Progress Check

1. Show how to declare a void function called count that has one long int pointer parameter called ptr.

2. When a pointer is passed to a function, can the function alter the contents of the object pointed to by the pointer?

3. Can an array be passed to a function? Explain.

CRITICAL SKILL 5.9: Returning Pointers

Functions can return pointers. Pointers are returned like any other data type and pose no special problem. However, because the pointer is one of C++'s more confusing features, a short discussion of pointer return types is warranted.

To return a pointer, a function must declare its return type to be a pointer. For example, here the return type of f( ) is declared to be an int pointer:

int *f();

If a function's return type is a pointer, then the value used in its return statement must also be a pointer. (As with all functions, the return value must be compatible with the return type.)

The following program demonstrates the use of a pointer return type. The function get_substr( ) searches a string for a substring. It returns a pointer to the first matching substring. If no match is found,

a null pointer is returned. For example, if the string is "I like C++" and the search string is "like", then the function returns a pointer to the l in "like".

```cpp
// Return a pointer.

#include <iostream>
using namespace std;

char *get_substr(char *sub, char *str);

int main()
{
  char *substr;

  substr = get_substr("three", "one two three four");

  cout << "substring found: " << substr;

  return 0;
}

// Return pointer to substring or null if not found.
char *get_substr(char *sub, char *str)
{
  int t;
  char *p, *p2, *start;

  for(t=0; str[t]; t++) {
    p = &str[t]; // reset pointers
    start = p;
    p2 = sub;
    while(*p2 && *p2==*p) { // check for substring
      p++;
      p2++;
    }

    /* If at end of p2 (i.e., substring), then

       a match has been found. */
    if(!*p2)
      return start; // return pointer to beginning of substring
  }
  return 0; // no match found
}
```
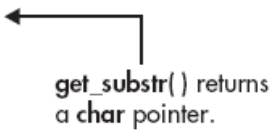
get_substr( ) returns a **char** pointer.

Here is the output produced by the program:

substring found: three four

The main( ) Function

As you know, the main( ) function is special because it is the first function called when your program executes. It signifies the beginning of your program. Unlike some programming languages that always begin execution at the "top" of the program, C++ begins every program with a call to the main( ) function, no matter where that function is located in the program. (However, it is common for main( ) to be the first function in your program so that it can be easily found.)

There can be only one main( ) in a program. If you try to include more than one, your program will not know where to begin execution. Actually, most compilers will catch this type of error and report it. As mentioned earlier, since main( ) is predefined by C++, it does not require a prototype.

CRITICAL SKILL 5.10: Pass Command-Line Arguments to main( )

Sometimes you will want to pass information into a program when you run it. This is generally accomplished by passing command-line arguments to main( ). A command-line argument is the information that follows the program's name on the command line of the operating system. (In Windows, the Run command also uses a command line.) For example, you might compile C++ programs from the command line by typing something like this:

cl prog-name

where prog-name is the program you want compiled. The name of the program is passed into the C++ compiler as a command-line argument.

C++ defines two built-in, but optional, parameters to main( ). They are argc and argv, and they receive the command-line arguments. These are the only parameters defined by C++ for main( ). However, other arguments may be supported in your specific operating environment, so you will want to check your compiler's documentation. Let's now look at argc and argv more closely.

NOTE : Technically, the names of the command-line parameters are arbitrary—you can use any names you like. However, **argc** and **argv** have been used by convention for several years, and it is best that you use these names so that anyone reading your program can quickly identify them as the command-line parameters.

The argc parameter is an integer that holds the number of arguments on the command line. It will always be at least 1, because the name of the program qualifies as the first argument.

The argv parameter is a pointer to an array of character pointers. Each pointer in the argv array points to a string containing a command-line argument. The program's name is pointed to by argv[0]; argv[1] will point to the first argument, argv[2] to the second argument, and so on. All command-line arguments are passed to the program as strings, so numeric arguments will have to be converted by your program into their proper internal format.

It is important that you declare argv properly. The most common method is

char *argv[];

You can access the individual arguments by indexing argv. The following program demonstrates how to access the command-line arguments. It displays all of the command-line arguments that are present when it is executed.

// Display command-line arguments.

Introducing Functions

```
// Display command-line arguments.

#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{

  for(int i = 0; i < argc; i++) {
    cout << argv[i] << "\n";
  }

  return 0;
}
```

For example, if the program is called ComLine, then executing it like this:

C>ComLine one two three

causes the following output:

ComLine
one
two
three

C++ does not stipulate the exact nature of a command-line argument, because host environments (operating systems) vary considerably on this point. However, the most common convention is as follows: each command-line argument must be separated by spaces or tabs. Often commas, semicolons, and the like are not valid argument separators. For example,

one, two, and three

is made up of four strings, while

one,two,and three

has two strings—the comma is not a legal separator.

If you need to pass a command-line argument that does, in fact, contain spaces, then you must place it between quotes. For example, this will be treated as a single command-line argument:

"this is one argument"

Keep in mind that the examples provided here apply to a wide variety of environments, but not necessarily to yours.

Usually, you will use argc and argv to get initial options or values (such as a filename) into your program. In C++, you can have as many command-line arguments as the operating system will allow. Using command-line arguments will give your program a professional appearance and facilitate the program's use in batch files.

Passing Numeric Command-Line Arguments

When you pass numeric data as a command-line argument to a program, that data will be received in string form. Your program will need to convert it into the binary, internal format using one of the standard library functions supported by C++. Three of the most commonly used functions for this purpose are shown here:

atof( )	Converts a string to a **double** and returns the result.
atol( )	Converts a string to a **long int** and returns the result.
atoi( )	Converts a string to an **int** and returns the result.

Each is called with a string containing a numeric value as an argument. Each uses the header <cstdlib>.

The following program demonstrates the conversion of a numeric command-line argument into its binary equivalent. It computes the sum of the two numbers that follow its name on the command line. The program uses the atof( ) function to convert its numeric argument into its internal representation.

```
/* This program displays the sum of the two numeric
   command line arguments.
*/

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

int main(int argc, char *argv[])
{

  double a, b;

  if(argc!=3) {
    cout << "Usage: add num num\n";
    return 1;
  }

  a = atof(argv[1]); // convert first command-line arg
  b = atof(argv[2]); // convert second comnand-line arg

  cout << a + b;

  return 0;
}
```

Use **atof**( ) to convert command-line numeric strings into **double**s.

To add two numbers, use this type of command line (assuming the program is called add):

C>add 100.2 231

Progress Check

1. What are the two parameters to main( ) usually called? Explain what each contains.

2. What is always the first command-line argument?

3. A numeric command-line argument is passed as string. True or false?

CRITICAL SKILL 5.11: Function Prototypes

Function prototypes were discussed briefly at the beginning of this module. Now it is time to explain them more fully. In C++, all functions must be declared before they are used. Typically, this is accomplished by use of a function prototype. Prototypes specify three things about a function:

- Its return type
- The type of its parameters
- The number of its parameters
- Prototypes allow the compiler to perform three important operations:

C++ A Beginner's Guide by Herbert Schildt

- They tell the compiler what type of code to generate when a function is called. Different return types must be handled differently by the compiler.
- They allow C++ to find and report any illegal type conversions between the type of arguments used to call a function and the type definition of its parameters.
- They allow the compiler to detect differences between the number of arguments used to call a function and the number of parameters in the function.

The general form of a function prototype is shown here. It is the same as a function definition, except that no body is present.

type func-name(type parm_name1, type parm_name2,..., type parm_nameN);

The use of parameter names in a prototype is optional. However, their use does let the compiler identify any type mismatches by name when an error occurs, so it is a good idea to include them.

To better understand the usefulness of function prototypes, consider the following program. If you try to compile it, an error message will be issued, because the program attempts to call sqr_it( ) with an integer argument instead of the integer pointer required. (There is no automatic conversion from integer to pointer.)

```
/* This program uses a function prototype to
   enforce strong type checking.
*/

void sqr_it(int *i); // prototype

int main()
{
  int x;

  x = 10;
  sqr_it(x); // Error! Type mismatch!

  return 0;
}

void sqr_it(int *i)
{
  *i = *i * *i;
}
```

Prototype prevents parameter type mismatches. Here, **sqr_it( )** is expecting a pointer but is called with an integer.

It is possible for a function definition to also serve as its prototype if the definition occurs prior to the function's first use in the program. For example, this is a valid program:

```
// Use a function's definition as its prototype.

#include <iostream>
using namespace std;

// Determine if a number is even.
bool isEven(int num) {
  if(!(num %2)) return true; // num is even
  return false;
}

int main()
{
  if(isEven(4)) cout << "4 is even\n";
  if(isEven(3)) cout << "this won't display";

  return 0;
}
```

Because **isEven( )** is defined before it is used, its definition also serves as its prototype.

Here, the function isEven( ) is defined before it is used in main( ). Thus, its definition can also serve as its prototype, and no separate prototype is needed.

In general, it is usually easier and better to simply declare a prototype for each function used by a program rather than trying to make sure that each function is defined before it is used. This is especially true for large programs in which it is hard to keep track of which functions use what other functions. Furthermore, it is possible to have two functions that call each other. In this case, prototypes must be used.

Headers Contain Prototypes

Earlier in this book, you were introduced to the standard C++ headers. You have learned that these headers contain information needed by your programs. While this partial explanation is true, it does not tell the whole story. C++'s headers contain the prototypes for the functions in the standard library. (They also contain various values and definitions used by those functions.) Like functions that you write, the standard library functions must be prototyped before they are used. For this reason, any program that uses a library function must also include the header containing the prototype of that function. To find out which header a library function requires, look in your compiler's library documentation. Along with a description of each function, you will find the name of the header that must be included in order to use that function.

Progress Check

1. What is a function prototype? What is the purpose of a prototype?

2. Aside from main( ), must all functions be prototyped?

3. When you use a standard library function, why must you include its header?

CRITICAL SKILL 5.12: Recursion

The last topic that we will examine in this module is recursion. Sometimes called circular definition, recursion is the process of defining something in terms of itself. As it relates to programming, recursion is the process of a function calling itself. A function that calls itself is said to be recursive.

The classic example of recursion is the function factr( ), which computes the factorial

of an integer. The factorial of a number N is the product of all the whole numbers between

1 and N. For example, 3 factorial is 1×2×3, or 6. Both factr( ) and its iterative equivalent

are shown here:

```
// Demonstrate recursion.

#include <iostream>
```

```cpp
using namespace std;

int factr(int n);
int fact(int n);

int main()
{
  // use recursive version
  cout << "4 factorial is " << factr(4);
  cout << '\n';

  // use iterative version
  cout << "4 factorial is " << fact(4);
  cout << '\n';

  return 0;
}

// Recursive version.
int factr(int n)
{
  int answer;

  if(n==1) return(1);
  answer = factr(n-1)*n;        ←———— Execute a recursive call to factr( ).
  return(answer);
}

// Iterative version.
int fact(int n)
{
  int t, answer;

  answer = 1;
  for(t=1; t<=n; t++) answer = answer*(t);
  return(answer);
}
```

The operation of the nonrecursive version of fact( ) should be clear. It uses a loop starting at 1 and progressively multiplies each number by the moving product.

The operation of the recursive factr( ) is a little more complex. When factr( ) is called with an argument of 1, the function returns 1; otherwise, it returns the product of factr(n–1)*n. To evaluate this expression, factr( ) is called with n–1. This happens until n equals 1 and the calls to the function begin returning. For example, when the factorial of 2 is calculated, the first call to factr( ) will cause a second call to be made with the argument of 1. This call will return 1, which is then multiplied by 2 (the original n value). The answer is then 2. You might find it interesting to insert cout statements into factr( ) that will show at what level each call is, and what the intermediate answers are.

C++ A Beginner's Guide by Herbert Schildt

When a function calls itself, new local variables and parameters are allocated storage (usually on the system stack), and the function code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point just after the recursive call inside the function. Recursive functions could be said to "telescope" out and back. Keep in mind that most recursive routines do not significantly reduce code size. Also, the recursive versions of most routines may execute a bit more slowly than their iterative equivalents, due to the added overhead of the additional function calls. Too many recursive calls to a function may cause a stack overrun. Because storage for function parameters and local variables is on the stack, and each new call creates a new copy of these variables, it is possible that the stack will be exhausted. If this occurs, other data may be destroyed as well. However, you probably will not have to worry about any of this unless a recursive function runs wild.

The main advantage of recursive functions is that they can be used to create versions of several algorithms that are clearer and simpler than those produced with their iterative relatives. For example, the Quicksort sorting algorithm is quite difficult to implement in an iterative way. Also, some problems, especially those related to artificial intelligence, seem to lend themselves to recursive solutions.

When writing a recursive function, you must include a conditional statement, such as an if, somewhere to force the function to return without execution of the recursive call. If you don't provide the conditional statement, then once you call the function, it will never return. This is a very common error. When developing programs with recursive functions, use cout statements liberally so that you can watch what is going on, and abort execution if you see that you have made a mistake. Here is another example of a recursive function, called reverse( ). It prints its string argument backwards on the screen.

```cpp
// Print a string backwards using recursion.

#include <iostream>
using namespace std;

void reverse(char *s);

int main()
{
  char str[] = "this is a test";

  reverse(str);
```

```
    return 0;
}

// Print string backwards.
void reverse(char *s)
{
  if(*s)
    reverse(s+1);
  else
    return;

  cout << *s;
}
```

The reverse( ) function first checks to see if it is passed a pointer to the null terminating the string. If not, then reverse( ) calls itself with a pointer to the next character in the string. When the null terminator is finally found, the calls begin unraveling, and the characters are displayed in reverse order.

One last point: Recursion is often difficult for beginners. Don't be discouraged if it seems a bit confusing right now. Over time, you will grow more accustomed to it.

Project 5-1 The Quicksort

In Module 4, you were shown a simple sorting method called the bubble sort.

QSDemo.cpp

It was mentioned at the time that substantially better sorts exist. Here you will develop a version of one of the best: the Quicksort. The Quicksort, invented and named by

C.A.R. Hoare, is the best general-purpose sorting algorithm currently available. The reason it could not be shown in Module 4 is that the best implementations of the Quicksort rely on recursion. The version we will develop sorts a character array, but the logic can be adapted to sort any type of object.

The Quicksort is built on the idea of partitions. The general procedure is to select a value, called the comparand, and then to partition the array into two sections. All elements greater than or equal to the comparand are put on one side, and those less than the value are put on the other. This process is then repeated for each remaining section until the array is sorted. For example, given the array fedacb and using the value d as the comparand, the first pass of the Quicksort would rearrange the array as follows:

This process is then repeated for each section—that is, bca and def. As you can see, the process is essentially recursive in nature and, indeed, the cleanest implementation of Quicksort is as a recursive function.

You can select the comparand value in two ways. You can either choose it at random, or you can select it by averaging a small set of values taken from the array. For optimal sorting, you should select a value that is precisely in the middle of the range of values. However, this is not easy to do for most sets of data. In the worst case, the value chosen is at one extremity.

Even in this case, however, Quicksort still performs correctly. The version of Quicksort that we will develop selects the middle element of the array as the comparand.

One other thing: The C++ library contains a function called qsort( ) which also performs a Quicksort. You might find it interesting to compare it to the version shown here.

Step By Step

1. Create a file called QSDemo.cpp.

2. The Quicksort will be implemented by a pair of functions. The first, called quicksort( ), provides a convenient interface for the user and sets up a call to the actual sorting function called qs( ). First, create the quicksort( ) function, as shown here:

```
// Set up a call to the actual sorting function.
void quicksort(char *items, int len)
{
  qs(items, 0, len-1);
}
```

Here, items points to the array to be sorted, and len specifies the number of elements in the array. As shown in the next step, qs( ) requires an initial partition, which quicksort( ) supplies. The advantage of using quicksort( ) is that it can be called with just a pointer to the array to be sorted and the number of elements in the array. It then provides the beginning and ending indices of the region to be sorted.

3. Add the actual Quicksort function, called qs( ), shown here:

```
// A recursive version of Quicksort for sorting characters.
void qs(char *items, int left, int right)
{
  int i, j;
  char x, y;

  i = left; j = right;
  x = items[( left+right) / 2 ];

  do {
```

C++ A Beginner's Guide by Herbert Schildt

```
    while((items[i] < x) && (i < right)) i++;
    while((x < items[j]) && (j > left)) j--;

    if(i <= j) {
      y = items[i];
      items[i] = items[j];
      items[j] = y;
      i++; j--;
    }
  } while(i <= j);

  if(left < j) qs(items, left, j);
  if(i < right) qs(items, i, right);
}
```

This function must be called with the indices of the region to be sorted. The left parameter must contain the beginning (left boundary) of the partition. The right parameter must contain the ending (right boundary) of the partition. When first called, the partition represents the entire array. Each recursive call progressively sorts a smaller partition.

4. To use the Quicksort, simply call quicksort( ) with the name of the array to be sorted and its length. After the call returns, the array will be sorted. Remember, this version works only for character arrays, but you can adapt the logic to sort any type of arrays you want.

5. Here is a program that demonstrates the Quicksort:

```
/*
   Project 5-1
   A version of the Quicksort for sorting characters.
*/

#include <iostream>
#include <cstring>

using namespace std;

void quicksort(char *items, int len);

void qs(char *items, int left, int right);

int main() {

  char str[] = "jfmckldoelazlkper";

  cout << "Original order: " << str << "\n";
```

```
   quicksort(str, strlen(str));

   cout << "Sorted order: " << str << "\n";

   return 0;

}

// Set up a call to the actual sorting function.
void quicksort(char *items, int len)
{
   qs(items, 0, len-1);
}

// A recursive version of Quicksort for sorting characters.
void qs(char *items, int left, int right)
{
   int i, j;
   char x, y;

   i = left; j = right;
   x = items[( left+right) / 2 ];

   do {
     while((items[i] < x) && (i < right)) i++;
     while((x < items[j]) && (j > left)) j--;

     if(i <= j) {
       y = items[i];
       items[i] = items[j];
       items[j] = y;
       i++; j--;
     }
   } while(i <= j);

   if(left < j) qs(items, left, j);
   if(i < right) qs(items, i, right);
}
```

The output from the program is shown here:

Original order: jfmckldoelazlkper

Sorted order: acdeefjkklllmoprz

Ask the Expert

Q: I have heard of something called the "default-to-int" rule. What is it and does it apply to C++?

A: In the original C language, and for early versions of C++, if no type specifier was present in a declaration, int was assumed. For example, in old-style code, the following function would be valid and would return an int result:

f() { // default to int return type

{ int x; // ... return x;

}

Here, the type returned by f( ) is int by default, since no other return type is specified. However, the "default-to-int" rule (also called the "implicit int" rule) is not supported by modern versions of C++. Although most compilers will continue to support the "default-to-int" rule for the sake of backward compatibility, you should explicitly specify the return type of every function that you write. Since older code frequently made use of the default integer return type, this change is also something to keep in mind when converting legacy code.

Module 5 Mastery Check

1. Show the general form of a function.
2. Create a function called hypot( ) that computes the length of the hypotenuse of a right triangle given the lengths of the two opposing sides. Demonstrate its use in a program. For this problem, you will need to use the sqrt( ) standard library function, which returns the square root of its argument. It has this prototype:

       double sqrt(double val);
       It uses the header <cmath>.

3. Can a function return a pointer? Can a function return an array?

4. Create your own version of the standard library function strlen( ). Call your version mystrlen( ), and demonstrate its use in a program.

5. Does a local variable maintain its value between calls to the function in which it is declared?

6. Give one benefit of global variables. Give one disadvantage.

7. Create a function called byThrees( ) that returns a series of numbers, with each value 3 greater than the preceding one. Have the series start at 0. Thus, the first five numbers returned by byThrees( ) are 0, 3, 6, 9, and 12. Create another function called reset( ) that causes byThrees( ) to start the series over again from 0. Demonstrate your functions in a program. Hint: You will need to use a global variable.

8. Write a program that requires a password that is specified on the command line. Your program doesn't have to actually do anything except report whether the password was entered correctly or incorrectly.

9. A prototype prevents a function from being called with the improper number of arguments. True or false?

10. Write a recursive function that prints the numbers 1 through 10. Demonstrate its use in a program.