Chapter 5

Functions: Mad Lib

Every program you've seen so far has consisted of one function: `main()`. However, once your programs reach a certain size or level of complexity, it becomes hard to work with them like this. Fortunately, there are ways to break up big programs into smaller, bite-sized chunks of code. In this chapter, you'll learn about one way—creating new functions. Specifically, you'll learn to:

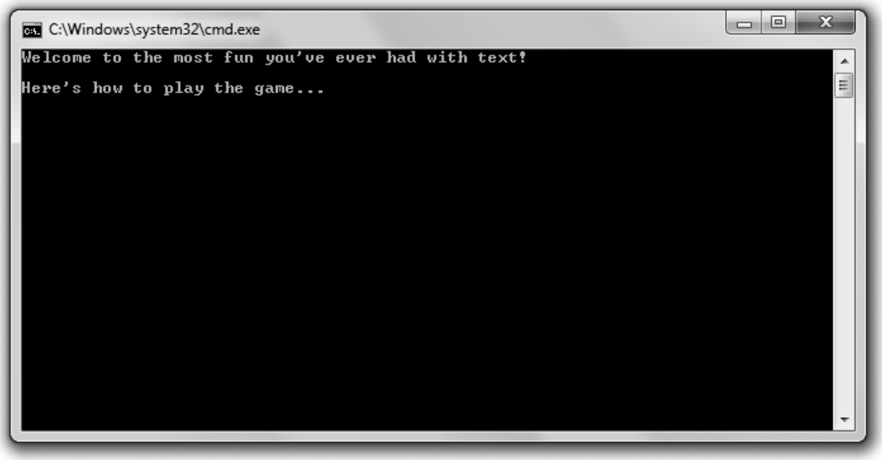
- Write new functions
- Accept values into your new functions through parameters
- Return information from your new functions through return values
- Work with global variables and constants
- Overload functions
- Inline functions

Creating Functions

C++ lets you write programs with multiple functions. Your new functions work just like the ones that are part of the standard language—they go off and perform a task and then return control to your program. A big advantage of writing new functions is that doing so allows you to break up your code into manageable pieces. Just like the functions you've already learned about from the standard library, your new functions should do one job well.

Introducing the Instructions Program

The results of the Instructions program are pretty basic—a few lines of text that are the beginning of some game instructions. From the looks of the output, Instructions seems like a program you could have written back in Chapter 1, "Types, Variables, and Standard I/O: Lost Fortune." But this program has a fresh element working behind the scenes—a new function. Take a look at Figure 5.1 to see the modest results of the code.



```
Welcome to the most fun you've ever had with text!

Here's how to play the game...
```

Figure 5.1
The instructions are displayed by a function.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 5 folder; the filename is `instructions.cpp`.

```
// Instructions
// Demonstrates writing new functions

#include <iostream>

using namespace std;
```

```
// function prototype (declaration)
void instructions();

int main()
{
    instructions();
    return 0;
}

// function definition
void instructions()
{
    cout << "Welcome to the most fun you've ever had with text!\n\n";
    cout << "Here's how to play the game...\n";
}
```

Declaring Functions

Before you can call a function you've written, you have to declare it. One way to declare a function is to write a *function prototype*—code that describes the function. You write a prototype by listing the return value of the function (or `void` if the function returns no value), followed by the name of the function, followed by a list of parameters between a set of parentheses. *Parameters* receive the values sent as arguments in a function call.

Just before the `main()` function, I write a function prototype:

```
void instructions();
```

In the preceding code, I declared a function named `instructions` that doesn't return a value. (You can tell this because I used `void` as the return type.) The function also takes no values, so it has no parameters. (You can tell this because there's nothing between the parentheses.)

Prototypes are not the only way to declare a function. Another way to accomplish the same thing is to let the function definition act as its own declaration. To do that, you simply have to put your function definition before the call to the function.

Hint

Although you don't have to use prototypes, they offer a lot of benefits—not the least of which is making your code clearer.

Defining Functions

Defining functions means writing all the code that makes the function tick. You define a function by listing the return value of the function (or `void` if the function returns no value), followed by the name of the function, followed by a list of parameters between a set of parentheses—just like a function prototype (except you don't end the line with a semicolon). This is called the *function header*. Then you create a block with curly braces that contains the instructions to be executed when the function is executed. This is called the *function body*.

At the end of the Instructions program, I define my simple `instructions()` function, which displays some game instructions. Because the function doesn't return any value, I don't need to use a `return` statement like I do in `main()`. I simply end the function definition with a closing curly brace.

```
void instructions()
{
    cout << "Welcome to the most fun you've ever had with text!\n\n";
    cout << "Here's how to play the game...\n";
}
```

Trap

A function definition must match its prototype on return type and function name; otherwise, you'll generate a compile error.

Calling Functions

You call your own functions the same way you call any other function—by writing the function's name followed by a pair of parentheses that encloses a valid list of arguments. In `main()`, I call my newly minted function simply with:

```
    instructions();
```

This line invokes `instructions()`. Whenever you call a function, control of the program jumps to that function. In this case, it means control jumps to `instructions()` and the program executes the function's code, which displays the game instructions. When a function finishes, control returns to the calling code. In this case, it means control returns to `main()`. The next statement in `main()` (`return 0;`) is executed and the program ends.

Understanding Abstraction

By writing and calling functions, you practice what's known as *abstraction.* Abstraction lets you think about the big picture without worrying about the details. In this program, I can simply use the function `instructions()` without worrying about the details of displaying the text. All I have to do is call the function with one line of code, and it gets the job done.

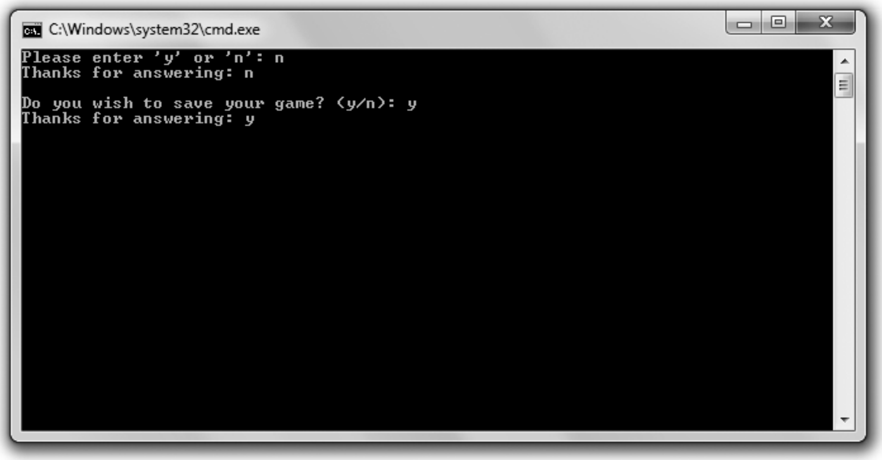
You might be surprised where you find abstraction, but people use it all the time. For example, consider two employees at a fast-food restaurant. If one tells the other that he just filled a Number 3 and "sized it," the other employee knows that the first employee took a customer's order, went to the heat lamps, grabbed a burger, went over to the deep fryer, filled their biggest cardboard container with french fries, went to the soda fountain, grabbed their biggest cup, filled it with soda, gave it all to the customer, took the customer's money, and gave the customer change. Not only would this level of detail make for a boring conversation, but also it's unnecessary. Both employees understand what it means to fill a Number 3 and "size it." They don't have to concern themselves with all the details because they're using abstraction.

Using Parameters and Return Values

As you've seen with standard library functions, you can provide a function value and get a value back. For example, with the `toupper()` function, you provide a character, and the function returns the uppercase version of it. Your own functions can also receive values and return a value. This allows your functions to communicate with the rest of your program.

Introducing the Yes or No Program

The Yes or No program asks the user typical questions a gamer might have to answer. First, the program asks the user to indicate yes or no. Then the program gets more specific and asks whether the user wants to save his game. Again, the results of the program are not remarkable; it's their implementation that's interesting. Each question is posed by a different function that communicates with `main()`. Figure 5.2 shows a sample run of the program.



```
Please enter 'y' or 'n': n
Thanks for answering: n

Do you wish to save your game? (y/n): y
Thanks for answering: y
```

Figure 5.2
Each question is asked by a separate function, and information is passed between these functions and `main()`.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 5 folder; the filename is `yes_or_no.cpp`.

```
// Yes or No
// Demonstrates return values and parameters

#include <iostream>
#include <string>

using namespace std;

char askYesNo1();
char askYesNo2(string question);

int main()
{
    char answer1 = askYesNo1();
    cout << "Thanks for answering: " << answer1 << "\n\n";

    char answer2 = askYesNo2("Do you wish to save your game?");
    cout << "Thanks for answering: " << answer2 << "\n";

    return 0;
}
```

```
char askYesNo1()
{
    char response1;
    do
    {
        cout << "Please enter 'y' or 'n': ";
        cin >> response1;
    } while (response1 != 'y' && response1 != 'n');

    return response1;
}

char askYesNo2(string question)
{
    char response2;
    do
    {
        cout << question << " (y/n): ";
        cin >> response2;
    } while (response2 != 'y' && response2 != 'n');

    return response2;
}
```

Returning a Value

You can return a value from a function to send information back to the calling code. To return a value, you need to specify a return type and then return a value of that type from the function.

Specifying a Return Type

The first function I declare, `askYesNo1()`, returns a `char` value. You can tell this from the function prototype before `main()`:

```
char askYesNo1();
```

You can also see this from the function definition after `main()`:

```
char askYesNo1()
```

Using the return Statement

`askYesNo1()` asks the user to enter y or n and keeps asking until he does. Once the user enters a valid character, the function wraps up with the following line, which returns the value of `response1`.

```
    return response1;
```

Notice that `response1` is a `char` value. It has to be because that's what I promised to return in both the function prototype and function definition.

A function ends whenever it hits a `return` statement. It's perfectly acceptable for a function to have more than one `return`. This just means that the function has several points at which it can end.

Trick

You don't have to return a value with a `return` statement. You can use `return` by itself in a function that returns no value (one that indicates `void` as its return type) to end the function.

Using a Returned Value

In `main()`, I call the function with the following line, which assigns the return value of the function to `answer1`.

```
    char answer1 = askYesNo1();
```

This means that `answer1` is assigned either `'y'` or `'n'`—whichever character the user entered when prompted by `askYesNo1()`.

Next, in `main()`, I display the value of `answer1` for all to see.

Accepting Values into Parameters

You can send a function values that it accepts into its parameters. This is the most common way to get information into a function.

Specifying Parameters

The second function I declare, `askYesNo2()`, accepts a value into a parameter. Specifically, it accepts a value of type `string`. You can tell this from the function prototype before `main()`:

```
char askYesNo2(string question);
```

Hint

You don't have to use parameter names in a prototype; all you have to include are the parameter types. For example, the following is a perfectly valid prototype which declares `askYesNo2()`, a function with one `string` parameter that returns a `char`.

```
char askYesNo2(string);
```

Even though you don't have to use parameter names in prototypes, it's a good idea to do so. It makes your code clearer, and it's worth the minor effort.

From the header of `askYesNo2()`, you can see that the function accepts a `string` object as a parameter and names that parameter `question`.

```
char askYesNo2(string question)
```

Unlike prototypes, you must specify parameter names in a function definition. You use a parameter name inside a function to access the parameter value.

Trap

The parameter types specified in a function prototype must match the parameter types listed in the function definition. If they don't, you'll generate a nasty compile error.

Passing Values to Parameters

The `askYesNo2()` function is an improvement over `askYesNo1()`. The new function allows you to ask your own personalized question by passing a string prompt to the function. In `main()`, I call `askYesNo2()` with:

```
char answer2 = askYesNo2("Do you wish to save your game?");
```

This statement calls `askYesNo2()` and passes the string literal argument `"Do you wish to save your game?"` to the function.

Using Parameter Values

`askYesNo2()` accepts `"Do you wish to save your game?"` into its parameter `question`, which acts like any other variable in the function. In fact, I display `question` with:

```
cout << question << " (y/n): ";
```

Hint

Actually, there's a little more going on behind the scenes here. When the string literal `"Do you wish to save your game?"` is passed to `question`, a `string` object equal to the string literal is created and the `string` object is assigned to `question`.

Just like `askYesNo1()`, `askYesNo2()` continues to prompt the user until he enters y or n. Then the function returns that value and ends.

Back in `main()`, the returned `char` value is assigned to `answer2`, which I then display.

Understanding Encapsulation

You might not see the need for return values when using your own functions. Why not just use the variables `response1` and `response2` back in the `main()`? Because you can't; `response1` and `response2` don't exist outside of the functions in which they were defined. In fact, no variable you create in a function, including its parameters, can be directly accessed outside its function. This is a good thing, and it is called *encapsulation*. Encapsulation helps keep independent code truly separate by hiding or encapsulating the details. That's why you use parameters and return values—to communicate only the information that needs to be exchanged. Plus, you don't have to keep track of variables you create within a function in the rest of your program. As your programs get large, this is a great benefit.

Encapsulation might sound a lot like abstraction. That's because they're closely related. Encapsulation is a principle of abstraction. Abstraction saves you from worrying about the details, while encapsulation hides the details from you. As an example, consider a television remote control with volume up and down buttons. When you use a TV remote to change the volume, you're employing abstraction because you don't need to know what happens inside the TV for it to work. Now suppose the TV remote has 10 volume levels. You can get to them all through the remote, but you can't directly access them. That is, you can't get a specific volume number directly. You can only press the up and down volume buttons to eventually get to the level you want. The actual volume number is encapsulated and not directly available to you.

Understanding Software Reuse

You can reuse functions in other programs. For example, since asking the user a yes or no question is such a common thing to do in a game, you could create an `askYesNo()` function and use it in all of your future game programs. So writing good functions not only saves you time and energy in your current game project, but it can save you effort in future ones, too.

In the Real World

It's always a waste of time to reinvent the wheel, so *software reuse*—employing existing software and other elements in new projects—is a technique that game companies take to heart. The benefits of software reuse include:

- **Increased company productivity.** By reusing code and other elements that already exist, such as a graphics engine, game companies can get their projects done with less effort.
- **Improved software quality.** If a game company already has a tested piece of code, such as a networking module, then the company can reuse the code with the knowledge that it's bug-free.
- **Improved software performance.** Once a game company has a high-performance piece of code, using it again not only saves the company the trouble of reinventing the wheel, it saves them from reinventing a less efficient one.

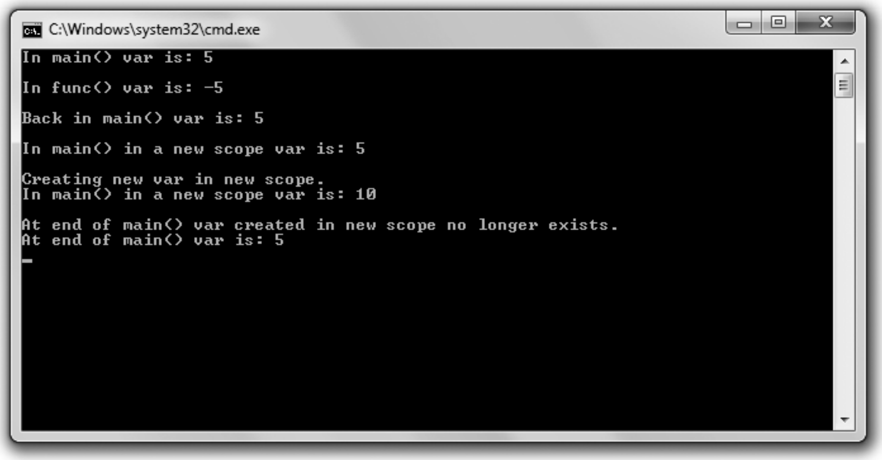
You can reuse code you've written by copying from one program and pasting it into another, but there is a better way. You can divide a big game project into multiple files. You'll learn about this technique in Chapter 10, "Inheritance and Polymorphism: Blackjack."

Working with Scopes

A variable's *scope* determines where the variable can be seen in your program. Scopes allow you to limit the accessibility of variables and are the key to encapsulation, helping keep separate parts of your program, such as functions, apart from each other.

Introducing the Scoping Program

The Scoping program demonstrates scopes. The program creates three variables with the same name in three separate scopes. The program displays the values of these variables, and you can see that even though they all have the same name, the variables are completely separate entities. Figure 5.3 shows the results of the program.



```
In main() var is: 5

In func() var is: -5

Back in main() var is: 5

In main() in a new scope var is: 5

Creating new var in new scope.
In main() in a new scope var is: 10

At end of main() var created in new scope no longer exists.
At end of main() var is: 5
_
```

Figure 5.3
Even though they have the same name, all three variables have a unique existence in their own scopes.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 5 folder; the filename is `scoping.cpp`.

```
// Scoping
// Demonstrates scopes

#include <iostream>

using namespace std;

void func();

int main()
{
    int var = 5;   // local variable in main()
    cout << "In main() var is: " << var << "\n\n";

    func();

    cout << "Back in main() var is: " << var << "\n\n";

    {
        cout << "In main() in a new scope var is: " << var << "\n\n";

        cout << "Creating new var in new scope.\n";
        int var = 10; // variable in new scope, hides other variable named var
        cout << "In main() in a new scope var is: " << var << "\n\n";
    }
    cout << "At end of main() var created in new scope no longer exists.\n";
    cout << "At end of main() var is: " << var << "\n";

    return 0;
}

void func()
{
    int var = -5; // local variable in func()
    cout << "In func() var is: " << var << "\n\n";
}
```

Working with Separate Scopes

Every time you use curly braces to create a block, you create a scope. Functions are one example of this. Variables declared in a scope aren't visible outside of that scope. This means that variables declared in a function aren't visible outside of that function.

Variables declared inside a function are considered *local variables*—they're local to the function. This is what makes functions encapsulated.

You've seen many local variables in action already. I define yet another local variable in `main()` with:

```
    int var = 5;    // local variable in main()
```

This line declares and initializes a local variable named `var`. I send the variable to `cout` in the next line of code:

```
    cout << "In main() var is: " << var << "\n\n";
```

This works just as you'd expect—5 is displayed.

Next, I call `func()`. Once I enter the function, I'm in a separate scope outside of the scope defined by `main()`. As a result, I can't access the variable `var` that I defined in `main()`. This means that when I next define a variable named `var` in `func()` with the following line, this new variable is completely separate from the variable named `var` in `main()`.

```
    int var = -5;    // local variable in func()
```

The two have no effect on each other, and that's the beauty of scopes. When you write a function, you don't have to worry if another function uses the same variable names.

Then, when I display the value of `var` in `func()` with the following line, the computer displays −5.

```
    cout << "In func() var is: " << var << "\n\n";
```

That's because, as far as the computer can see in this scope, there's only one variable named var—the local variable I declared in this function.

Once a scope ends, all of the variables declared in that scope cease to exist. They're said to go *out of scope*. So next, when func() ends, its scope ends. This means all of the variables declared in func() are destroyed. As a result, the var I declared in func() with a value of −5 is destroyed.

After func() ends, control returns to main() and picks up right where it left off. Next, the following line is executed, which sends var to cout.

```
cout << "Back in main() var is: " << var << "\n\n";
```

The value of the var local to main() (5) is displayed again.

You might be wondering what happened to the var I created in main() while I was in func(). Well, the variable wasn't destroyed because main() hadn't yet ended. (Program control simply took a small detour to func().) When a program momentarily exits one function to enter another, the computer saves its place in the first function, keeping safe the values of all of its local variables, which are reinstated when control returns to the first function.

Hint

Parameters act just like local variables in functions.

Working with Nested Scopes

You can create a nested scope with a pair of curly braces in an existing scope. That's what I do next in main(), with:

```
{
    cout << "In main() in a new scope var is: " << var << "\n\n";

    cout << "Creating new var in new scope.\n";
    int var = 10;   // variable in new scope, hides other variable named var
    cout << "In main() in a new scope var is: " << var << "\n\n";
}
```

This new scope is a nested scope in main(). The first thing I do in this nested scope is display var. If a variable hasn't been declared in a scope, the computer looks up the levels of nested scopes one at a time to find the variable you requested. In this case, because var hasn't been declared in this nested scope, the computer looks one level up to the scope that defines main() and finds var. As a result, the program displays that variable's value—5.

However, the next thing I do in this nested scope is declare a new variable named `var` and initialize it to 10. Now when I send `var` to `cout`, 10 is displayed. This time the computer doesn't have to look up any levels of nested scopes to find `var`; there's a `var` local to this scope. And don't worry, the `var` I first declared in `main()` still exists; it's simply hidden in this nested scope by the new `var`.

Trap

Although you can declare variables with the same name in a series of nested scopes, it's not a good idea because it can lead to confusion.

Next, when the nested scope ends, the `var` that was equal to 10 goes out of scope and ceases to exist. However, the first `var` I created is still around, so when I display `var` for the last time in `main()` with the following line, the program displays 5.

```
cout << "At end of main() var is: " << var << "\n";
```

Hint

When you define variables inside `for` loops, `while` loops, `if` statements, and `switch` statements, these variables don't exist outside their structures. They act like variables declared in a nested scope. For example, in the following code, the variable `i` doesn't exist outside the loop.

```
for(int i = 0; i < 10; ++i)
{
    cout << i;
}
// i doesn't exist outside the loop
```

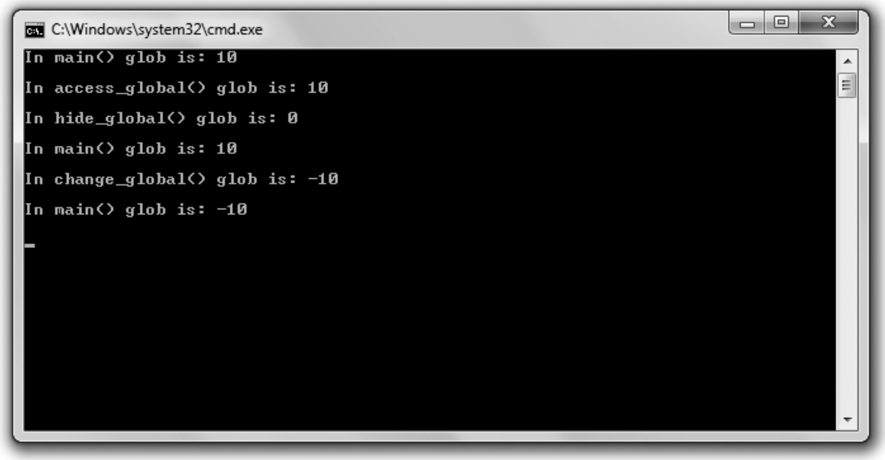
But beware—some older compilers don't properly implement this functionality of standard C++. I recommend that you use an IDE with a modern compiler, such as Microsoft Visual Studio Express 2013 for Windows Desktop. For step-by-step instructions on how to create your first project with this IDE, check out Appendix A, "Creating Your First C++ Program."

Using Global Variables

Through the magic of encapsulation, the functions you've seen are all totally sealed off and independent from each other. The only way to get information into them is through their parameters, and the only way to get information out of them is from their return values. Well, that's not completely true. There is another way to share information among parts of your program—through *global variables* (variables that are accessible from any part of your program).

Introducing the Global Reach Program

The Global Reach program demonstrates global variables. The program shows how you can access a global variable from anywhere in your program. It also shows how you can hide a global variable in a scope. Finally, it shows that you can change a global variable from anywhere in your program. Figure 5.4 shows the results of the program.



```
In main() glob is: 10
In access_global() glob is: 10
In hide_global() glob is: 0
In main() glob is: 10
In change_global() glob is: -10
In main() glob is: -10
```

Figure 5.4
You can access and change global variables from anywhere in a program—but they can be hidden in a scope as well.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 5 folder; the filename is `global_reach.cpp`.

```
// Global Reach
// Demonstrates global variables

#include <iostream>

using namespace std;

int glob = 10; // global variable

void access_global();
void hide_global();
void change_global();
```

```
int main()
{
    cout << "In main() glob is: " << glob << "\n\n";
    access_global();

    hide_global();
    cout << "In main() glob is: " << glob << "\n\n";

    change_global();
    cout << "In main() glob is: " << glob << "\n\n";

    return 0;
}

void access_global()
{
    cout << "In access_global() glob is: " << glob << "\n\n";
}

void hide_global()
{
    int glob = 0;   // hide global variable glob
    cout << "In hide_global() glob is: " << glob << "\n\n";
}

void change_global()
{
    glob = -10;   // change global variable glob
    cout << "In change_global() glob is: " << glob << "\n\n";
}
```

Declaring Global Variables

You declare global variables outside of any function in your program file. That's what I do in the following line, which creates a global variable named `glob` initialized to 10.

```
int glob = 10;   // global variable
```

Accessing Global Variables

You can access a global variable from anywhere in your program. To prove it, I display `glob` in `main()` with:

```
    cout << "In main() glob is: " << glob << "\n\n";
```

The program displays 10 because as a global variable, `glob` is available to any part of the program. To show this again, I next call `access_global()`, and the computer executes the following code in that function:

```
cout << "In access_global() glob is: " << glob << "\n\n";
```

Again, 10 is displayed. That makes sense because I'm displaying the exact same variable in each function.

Hiding Global Variables

You can hide a global variable like any other variable in a scope; you simply declare a new variable with the same name. That's exactly what I do next, when I call `hide_global()`. The key line in that function doesn't change the global variable `glob`; instead, it creates a new variable named `glob`, local to `hide_global()`, that hides the global variable.

```
int glob = 0;  // hide global variable glob
```

As a result, when I send `glob` to `cout` next in `hide_global()` with the following line, 0 is displayed.

```
cout << "In hide_global() glob is: " << glob << "\n\n";
```

The global variable `glob` remains hidden in the scope of `hide_global()` until the function ends.

To prove that the global variable was only hidden and not changed, next I display `glob` back in `main()` with:

```
cout << "In main() glob is: " << glob << "\n\n";
```

Once again, 10 is displayed.

Trap

Although you can declare variables in a function with the same name as a global variable, it's not a good idea because it can lead to confusion.

Altering Global Variables

Just as you can access a global variable from anywhere in your program, you can alter one from anywhere in your program, too. That's what I do next, when I call the `change_global()` function. The key line of the function assigns –10 to the global variable `glob`.

```
glob = -10; // change global variable glob
```

To show that it worked, I display the variable in `change_global()` with:

```
    cout << "In change_global() glob is: " << glob << "\n\n";
```

Then, back in `main()`, I send `glob` to `cout` with:

```
    cout << "In main() glob is: " << glob << "\n\n";
```

Because the global variable `glob` was changed, −10 is displayed.

Minimizing the Use of Global Variables

Just because you can doesn't mean you should. This is a good programming motto. Sometimes things are technically possible but not a good idea. Using global variables is an example of this. In general, global variables make programs confusing because it can be difficult to keep track of their changing values. You should limit your use of global variables as much as possible.

Using Global Constants

Unlike global variables, which can make your programs confusing, *global constants*—constants that can be accessed from anywhere in your program—can help make programs clearer. You declare a global constant much like you declare a global variable—by declaring it outside of any function. And because you're declaring a constant, you need to use the `const` keyword. For example, the following line defines a global constant (assuming the declaration is outside of any function) named `MAX_ENEMIES` with a value of 10 that can be accessed anywhere in the program.

```
const int MAX_ENEMIES = 10;
```

Trap

Just like with global variables, you can hide a global constant by declaring a local constant with the same name. However, you should avoid this because it can lead to confusion.

How exactly can global constants make game programming code clearer? Well, suppose you're writing an action game in which you want to limit the total number of enemies that can blast the poor player at once. Instead of using a numeric literal everywhere, such as 10, you could define a global constant `MAX_ENEMIES` that's equal to 10. Then whenever you see that global constant name, you know exactly what it stands for.

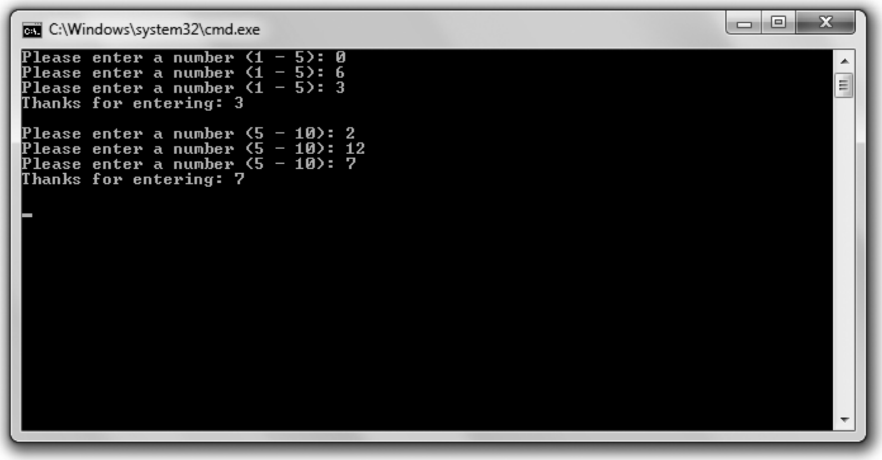
One caveat: You should only use global constants if you need a constant value in more than one part of your program. If you only need a constant value in a specific scope (such as in a single function), use a local constant instead.

Using Default Arguments

When you write a function in which a parameter almost always gets passed the same value, you can save the caller the effort of constantly specifying this value by using a *default argument*—a value assigned to a parameter if none is specified. Here's a concrete example. Suppose you have a function that sets the graphics display. One of your parameters might be `bool fullScreen`, which tells the function whether to display the game in full screen or windowed mode. Now, if you think the function will often be called with `true` for `fullScreen`, you could give that parameter a default argument of `true`, saving the caller the effort of passing `true` to `fullScreen` whenever the caller invokes this display-setting function.

Introducing the Give Me a Number Program

The Give Me a Number program asks the user for two different numbers in two different ranges. The same function is called each time the user is prompted for a number. However, each call to this function uses a different number of arguments because this function has a default argument for the lower limit. This means the caller can omit an argument for the lower limit, and the function will use a default value automatically. Figure 5.5 shows the results of the program.


C:\Windows\system32\cmd.exe

```
Please enter a number (1 - 5): 0
Please enter a number (1 - 5): 6
Please enter a number (1 - 5): 3
Thanks for entering: 3

Please enter a number (5 - 10): 2
Please enter a number (5 - 10): 12
Please enter a number (5 - 10): 7
Thanks for entering: 7

_
```

Figure 5.5
A default argument is used for the lower limit the first time the user is prompted for a number.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 5 folder; the filename is `give_me_a_number.cpp`.

```
// Give Me a Number
// Demonstrates default function arguments

#include <iostream>
#include <string>

using namespace std;

int askNumber(int high, int low = 1);

int main()
{
    int number = askNumber(5);
    cout << "Thanks for entering: " << number << "\n\n";

    number = askNumber(10, 5);
    cout << "Thanks for entering: " << number << "\n\n";

    return 0;
}

int askNumber(int high, int low)
{
    int num;
    do
    {
        cout << "Please enter a number" << " (" << low << " - " << high << "): ";
        cin >> num;
    } while (num > high || num < low);

    return num;
}
```

Specifying Default Arguments

The function `askNumber()` has two parameters: `high` and `low`. You can tell this from the function prototype:

```
int askNumber(int high, int low = 1);
```

Notice that the second parameter, `low`, looks like it's assigned a value. In a way, it is. The `1` is a default argument, meaning that if a value isn't passed to `low` when the function is called, `low` is assigned `1`. You specify default arguments by using = followed by a value after a parameter name.

Trap

Once you specify a default argument in a list of parameters, you must specify default arguments for all remaining parameters. So the following prototype is valid:

```
void setDisplay(int height, int width, int depth = 32, bool fullScreen = true);
```

while this one is illegal:

```
void setDisplay(int width, int height, int depth = 32, bool fullScreen);
```

By the way, you don't repeat the default argument in the function definition, as you can see in the function definition of `askNumber()`.

```
int askNumber(int high, int low)
```

Assigning Default Arguments to Parameters

The `askNumber()` function asks the user for a number between an upper and a lower limit. The function keeps asking until the user enters a number within the range, and then it returns the number. I first call the function in `main()` with:

```
int number = askNumber(5);
```

As a result of this code, the parameter `high` in `askNumber()` is assigned 5. Because I don't provide any value for the second parameter, `low`, it is assigned the default value of 1. This means the function prompts the user for a number between 1 and 5.

Trap

When you are calling a function with default arguments, once you omit an argument, you must omit arguments for all remaining parameters. For example, given the prototype

```
void setDisplay(int height, int width, int depth = 32, bool fullScreen = true);
```

a valid call to the function would be

```
setDisplay(1680, 1050);
```

while an illegal call would be

```
setDisplay(1680, 1050, false);
```

Once the user enters a valid number, `askNumber()` returns that value and ends. Back in `main()`, the value is assigned to `number` and displayed.

Overriding Default Arguments

Next, I call `askNumber()` again with:

```
    number = askNumber(10, 5);
```

This time I pass a value for `low`—5. This is perfectly fine; you can pass an argument for any parameter with a default argument, and the value you pass will override the default. In this case, it means that `low` is assigned 5.

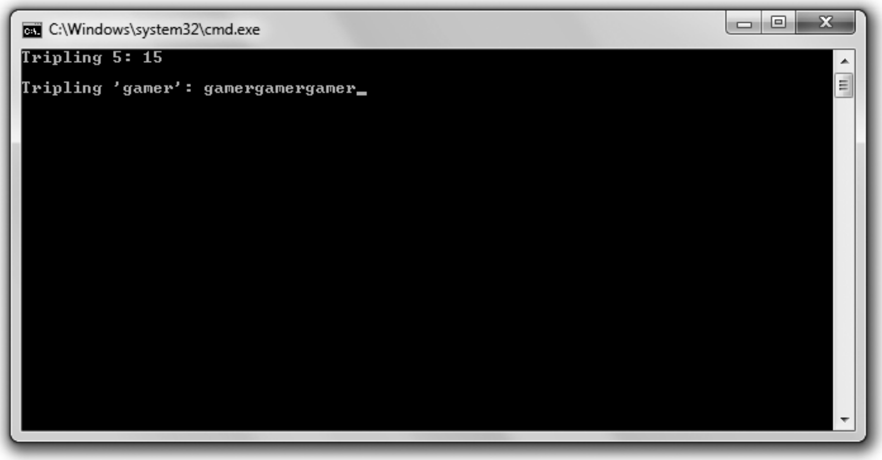
As a result, the user is prompted for a number between 5 and 10. Once the user enters a valid number, `askNumber()` returns that value and ends. Back in `main()`, the value is assigned to `number` and displayed.

Overloading Functions

You've seen how you must specify a parameter list and a single return type for each function you write. But what if you want a function that's more versatile—one that can accept different sets of arguments? For example, suppose you want to write a function that performs a 3D transformation on a set of vertices that are represented as `floats`, but you want the function to work with `ints` as well. Instead of writing two separate functions with two different names, you could use function overloading so that a single function could handle the different parameter lists. This way, you could call one function and pass vertices as either `floats` or `ints`.

Introducing the Triple Program

The Triple program triples the value 5 and "gamer". The program triples these values using a single function that's been overloaded to work with an argument of two different types: `int` and `string`. Figure 5.6 shows a sample run of the program.



```
Tripling 5: 15

Tripling 'gamer': gamergamergamer_
```

Figure 5.6
Function overloading allows you to triple the values of two different types using the same function name.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 5 folder; the filename is `triple.cpp`.

```
// Triple
// Demonstrates function overloading

#include <iostream>
#include <string>

using namespace std;

int triple(int number);
string triple(string text);

int main()
{
    cout << "Tripling 5: " << triple(5) << "\n\n";
    cout << "Tripling 'gamer': " << triple("gamer");

    return 0;
}

int triple(int number)
{
    return (number * 3);
}
```

```
string triple(string text)
{
    return (text + text + text);
}
```

Creating Overloaded Functions

To create an overloaded function, you simply need to write multiple function definitions with the same name and different parameter lists. In the Triple program, I write two definitions for the function `triple()`, each of which specifies a different type as its single argument. Here are the function prototypes:

```
int triple(int number);
string triple(string text);
```

The first takes an `int` argument and returns an `int`. The second takes a `string` object and returns a `string` object.

In each function definition, you can see that I return triple the value sent. In the first function, I return the `int` sent, tripled. In the second function, I return the string sent, repeated three times.

Trap

To implement function overloading, you need to write multiple definitions for the same function with different parameter lists. Notice that I didn't mention anything about return types. That's because if you write two function definitions in which only the return type is different, you'll generate a compile error. For example, you cannot have both of the following prototypes in a program:

```
int Bonus(int);
float Bonus(int);
```

Calling Overloaded Functions

You can call an overloaded function the same way you call any other function, by using its name with a set of valid arguments. But with overloaded functions, the compiler (based on the argument values) determines which definition to invoke. For example, when I call `triple()` with the following line and use an `int` as the argument, the compiler knows to invoke the definition that takes an `int`. As a result, the function returns the `int` 15.

```
    cout << "Tripling 5: " << triple(5) << "\n\n";
```

I call `triple()` again with:

```
cout << "Tripling 'gamer': " << triple("gamer");
```

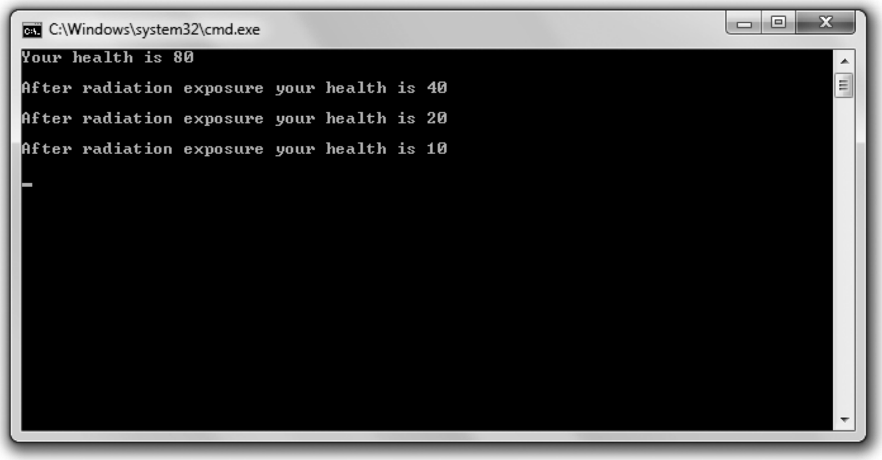
Because I use a string literal as the argument, the compiler knows to invoke the definition of the function that takes a `string` object. As a result, the function returns the `string` object equal to gamergamergamer.

Inlining Functions

There's a small performance cost associated with calling a function. Normally this isn't a big deal because the cost is relatively minor. However, for tiny functions (such as one or two lines), it's sometimes possible to speed up program performance by inlining them. By *inlining* a function, you ask the compiler to make a copy of the function everywhere it's called. As a result, program control doesn't have to jump to a different location each time the function is called.

Introducing the Taking Damage Program

The Taking Damage program simulates what happens to a character's health as the character takes radiation damage. The character loses half of his health each round. Fortunately, the program runs only three rounds, so we're spared the sad end of the character. The program inlines the tiny function that calculates the character's new health. Figure 5.7 shows the program results.



```
Your health is 80

After radiation exposure your health is 40

After radiation exposure your health is 20

After radiation exposure your health is 10

_
```

Figure 5.7
The character approaches his demise quite efficiently as his health decreases through an inlined function.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 5 folder; the filename is `taking_damage.cpp`.

```
// Taking Damage
// Demonstrates function inlining

#include <iostream>

int radiation(int health);

using namespace std;

int main()
{
    int health = 80;
    cout << "Your health is " << health << "\n\n";

    health = radiation(health);
    cout << "After radiation exposure your health is " << health << "\n\n";

    health = radiation(health);
    cout << "After radiation exposure your health is " << health << "\n\n";

    health = radiation(health);
    cout << "After radiation exposure your health is " << health << "\n\n";

    return 0;
}

inline int radiation(int health)
{
    return (health / 2);
}
```

Specifying Functions for Inlining

To mark a function for inlining, simply put `inline` before the function definition. That's what I do when I define the following function:

```
inline int radiation(int health)
```

Note that you don't use `inline` in the function declaration:

```
int radiation(int health);
```

By flagging the function with `inline`, you ask the compiler to copy the function directly into the calling code. This saves the overhead of making the function call. That is, program control doesn't have to jump to another part of your code. For small functions, this can result in a performance boost.

However, inlining is not a silver bullet for performance. In fact, indiscriminate inlining can lead to worse performance because inlining a function creates extra copies of it, which can dramatically increase memory consumption.

Hint

When you inline a function, you really make a request to the compiler, which has the ultimate decision on whether to inline the function. If your compiler thinks that inlining won't boost performance, it won't inline the function.

Calling Inlined Functions

Calling an inlined function is no different than calling a non-inlined function, as you see with my first call to `radiation()`.

```
health = radiation(health);
```

This line of code assigns `health` one-half of its original value.

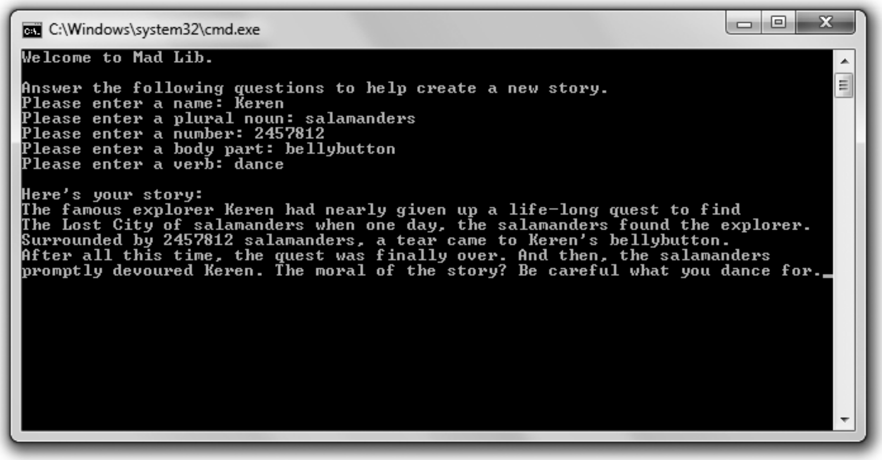
Assuming that the compiler grants my request for inlining, this code doesn't result in a function call. Instead, the compiler places the code to halve `health` right at this place in the program. In fact, the compiler does this for all three calls to the function.

In the Real World

Although obsessing about performance is a game programmer's favorite hobby, there's a danger in focusing too much on speed. In fact, the approach many developers take is to first get their game programs working well before they tweak for small performance gains. At that point, programmers will *profile* their code by running a utility (a profiler) that analyzes where the game program spends its time. If a programmer sees bottlenecks, he or she might consider hand optimizations such as function inlining.

Introducing the Mad Lib Game

The Mad Lib game asks for the user's help in creating a story. The user supplies the name of a person, a plural noun, a number, a body part, and a verb. The program takes all of this information and uses it to create a personalized story. Figure 5.8 shows a sample run of the program.



```
Welcome to Mad Lib.

Answer the following questions to help create a new story.
Please enter a name: Keren
Please enter a plural noun: salamanders
Please enter a number: 2457812
Please enter a body part: bellybutton
Please enter a verb: dance

Here's your story:
The famous explorer Keren had nearly given up a life-long quest to find
The Lost City of salamanders when one day, the salamanders found the explorer.
Surrounded by 2457812 salamanders, a tear came to Keren's bellybutton.
After all this time, the quest was finally over. And then, the salamanders
promptly devoured Keren. The moral of the story? Be careful what you dance for._
```

Figure 5.8
After the user provides all of the necessary information, the program displays the literary masterpiece.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 5 folder; the filename is `mad_lib.cpp`.

Setting Up the Program

As usual, I start the program with some comments and include the necessary files.

```
// Mad-Lib
// Creates a story based on user input

#include <iostream>
#include <string>

using namespace std;

string askText(string prompt);
int askNumber(string prompt);
void tellStory(string name, string noun, int number, string bodyPart, string verb);
```

You can tell from my function prototypes that I have three functions in addition to `main()`—`askText()`, `askNumber()`, and `tellStory()`.

The main( ) Function

The `main()` function calls all of the other functions. It calls the function `askText()` to get a name, plural noun, body part, and verb from the user. It calls `askNumber()` to get a number from the user. It calls `tellStory()` with all of the user-supplied information to generate and display the story.

```
int main()
{
    cout << "Welcome to Mad Lib.\n\n";
    cout << "Answer the following questions to help create a new story.\n";

    string name = askText("Please enter a name: ");
    string noun = askText("Please enter a plural noun: ");
    int number = askNumber("Please enter a number: ");
    string bodyPart = askText("Please enter a body part: ");
    string verb = askText("Please enter a verb: ");

    tellStory(name, noun, number, bodyPart, verb);

    return 0;
}
```

The askText( ) Function

The `askText ()` function gets a string from the user. The function is versatile and takes a parameter of type `string`, which it uses to prompt the user. Because of this, I'm able to call this single function to ask the user for a variety of different pieces of information, including a name, plural noun, body part, and verb.

```
string askText(string prompt)
{
    string text;
    cout << prompt;
    cin >> text;
    return text;
}
```

Trap

Remember that this simple use of `cin` works only with strings that have no white space in them (such as tabs or spaces). So when a user is prompted for a body part, he can enter `bellybutton`, but `medulla oblongata` will cause a problem for the program.

There are ways to compensate for this, but that really requires a discussion of something called *streams*, which is beyond the scope of this book. So use `cin` in this way, but just be aware of its limitations.

The askNumber( ) Function

The `askNumber()` function gets an integer from the user. Although I only call it once in the program, it's versatile because it takes a parameter of type `string` that it uses to prompt the user.

```
int askNumber(string prompt)
{
    int num;
    cout << prompt;
    cin >> num;
    return num;
}
```

The tellStory( ) Function

The `tellStory()` function takes all of the information entered by the user and uses it to display a personalized story.

```
void tellStory(string name, string noun, int number, string bodyPart, string verb)
{
    cout << "\nHere's your story:\n";
    cout << "The famous explorer ";
    cout << name;
    cout << " had nearly given up a life-long quest to find\n";
    cout << "The Lost City of ";
    cout << noun;
    cout << " when one day, the ";
    cout << noun;
    cout << " found the explorer.\n";
    cout << "Surrounded by ";
    cout << number;
    cout << " " << noun;
    cout << ", a tear came to ";
    cout << name << "'s ";
    cout << bodyPart << ".\n";
    cout << "After all this time, the quest was finally over. ";
    cout << "And then, the ";
    cout << noun << "\n";
    cout << "promptly devoured ";
    cout << name << ". ";
    cout << "The moral of the story? Be careful what you ";
    cout << verb;
    cout << " for.";
}
```

Summary

In this chapter, you should have learned the following concepts:

- Functions allow you to break up your programs into manageable chunks.
- One way to declare a function is to write a function prototype—code that lists the return value, name, and parameter types of a function.
- Defining a function means writing all the code that makes the function tick.
- You can use the `return` statement to return a value from a function. You can also use `return` to end a function that has `void` as its return type.
- A variable's scope determines where the variable can be seen in your program.
- Global variables are accessible from any part of your program. In general, you should try to limit your use of global variables.
- Global constants are accessible from any part of your program. Using global constants can make your program code clearer.
- Default arguments are assigned to a parameter if no value for the parameter is specified in the function call.
- Function overloading is the process of creating multiple definitions for the same function, each of which has a different set of parameters.
- Function inlining is the process of asking the compiler to inline a function—meaning that the compiler should make a copy of the function everywhere in the code where the function is called. Inlining very small functions can sometimes yield a performance boost.

Questions and Answers

Q: Why should I write functions?

A: Functions allow you to break up your programs into logical pieces. These pieces result in smaller, more manageable chunks of code, which are easier to work with than a single monolithic program.

Q: What's encapsulation?
A: At its core, encapsulation is about keeping things separate. Function encapsulation provides that variables declared in a function are not accessible outside the function, for example.

Q: What's the difference between an argument and a parameter?
A: An argument is what you use in a function call to pass a value to a function. A parameter is what you use in a function definition to accept values passed to a function.

Q: Can I have more than one `return` statement in a function?
A: Sure. In fact, you might want multiple `return` statements to specify different end points of a function.

Q: What's a local variable?
A: A variable that's defined in a scope. All variables defined in a function are local variables; they're local to that function.

Q: What does it mean to hide a variable?
A: A variable is hidden when you declare it inside a new scope with the same name as a variable in an outer scope. As a result, you can't get to the variable in the outer scope by using its variable name in the inner scope.

Q: When does a variable go out of scope?
A: A variable goes out of scope when the scope in which it was created ends.

Q: What does it mean when a variable goes out of scope?
A: It means the variable ceases to exist.

Q: What's a nested scope?
A: A scope created within an existing scope.

Q: Must an argument have the same name as the parameter to which it's passed?
A: No. You're free to use different names. Only the value is passed from a function call to a function.

Q: Can I write one function that calls another?
A: Of course. In fact, whenever you write a function that you call from `main()`, you're doing just that. In addition, you can write a function (other than `main()`) that calls another function.

Q: What is code profiling?
A: It's the process of recording how much CPU time various parts of a program use.

Q: Why profile code?
A: To determine any bottlenecks in a program. Sometimes it makes sense to revisit these sections of code in an attempt to optimize them.

Q: When do programmers profile code?
A: Usually toward the end of the programming of a game project.

Q: What is premature optimization?
A: An attempt to optimize code too early in the development process. Code optimization usually makes sense near the end of programming a game project.

Discussion Questions

1. How does function encapsulation help you write better programs?
2. How can global variables make code confusing?
3. How can global constants make code clearer?
4. What are the pros and cons of optimizing code?
5. How can software reuse benefit the game industry?

Exercises

1. What's wrong with the following prototype?

   ```
   int askNumber(int low = 1, int high);
   ```

2. Rewrite the Hangman game from Chapter 4, "The Standard Template Library: Hangman," using functions. Include a function to get the player's guess and another function to determine whether the player's guess is in the secret word.
3. Using default arguments, write a function that asks the user for a number and returns that number. The function should accept a string prompt from the calling code. If the caller doesn't supply a string for the prompt, the function should use a generic prompt. Next, using function overloading, write a function that achieves the same results.