Procedural Abstraction and Functions That Return a Value

Procedural Abstraction and Functions That Return a Value

There was a most ingenious Architect who had contrived a new method for building Houses, by beginning at the Roof, and working downward to the Foundation.

JONATHAN SWIFT, *GULLIVER'S TRAVELS*

Introduction

A program can be thought of as consisting of subparts, such as obtaining the input data, calculating the output data, and displaying the output data. C++, like most programming languages, has facilities to name and code each of these subparts separately. In C++ these subparts are called **functions.** In this chapter we present the basic syntax for one of the two main kinds of C++ functions—namely those designed to compute a single value. We also discuss how these functions can aid in program design. We begin with a discussion of a fundamental design principle.

Prerequisites

You should read Chapter 2 and at least look through Chapter 1 before reading this chapter.

3.1 Top-Down Design

Remember that the way to write a program is to first design the method that the program will use and to write out this method in English, as if the instructions were to be followed by a human clerk. As we noted in Chapter 1, this set of instructions is called an *algorithm*. A good plan of attack for designing the algorithm is to break down the task to be accomplished into a few subtasks, decompose each of these subtasks into smaller subtasks, and so forth. Eventually the subtasks become so small that they are trivial to implement in C++. This method is called **top-down design.** (The method is also sometimes called **stepwise refinement,** or more graphically, **divide and conquer.**)

Using the top-down method, you design a program by breaking the program's task into subtasks and solving these subtasks by subalgorithms. Preserving this top-down structure in your C++ program would make the program easier to understand,

easier to change if need be, and as will become apparent, easier to write, test, and debug. C++, like most programming languages, has facilities to include separate subparts inside of a program. In other programming languages these subparts are called *subprograms* or *procedures*. In C++ these subparts are called *functions*.

functions for teamwork

One of the advantages of using functions to divide a programming task into subtasks is that different people can work on the different subtasks. When producing a very large program, such as a compiler or office-management system, this sort of teamwork is needed if the program is to be produced in a reasonable amount of time. We will begin our discussion of functions by showing you how to use functions that were written by somebody else.

3.2 Predefined Functions

C++ comes with libraries of predefined functions that you can use in your programs. Before we show you how to define functions, we will first show you how to use these functions that are already defined for you.

Using Predefined Functions

We will use the `sqrt` function to illustrate how you use predefined functions. The `sqrt` function calculates the square root of a number. (The square root of a number is that number which, when multiplied by itself, will produce the number you started out with. For example, the square root of 9 is 3 because 3^2 is equal to 9.) The function `sqrt` starts with a number, such as 9.0, and computes its square root, in this case 3.0. The value the function starts out with is called its **argument.** The value it computes is called the **value returned.** Some functions may have more than one argument, but no function has more than one value returned. If you think of the function as being similar to a small program, then the arguments are analogous to the input and the value returned is analogous to the output.

argument
value returned

The syntax for using functions in your program is simple. To set a variable named `the_root` equal to the square root of `9.0`, you can use the following assignment statement:

```
the_root = sqrt(9.0);
```

The expression `sqrt(9.0)` is called a **function call** (or if you want to be fancy you can also call it a **function invocation**). An argument in a function call can be a constant, such as `9.0`, or a variable, or a more complicated expression. A function call is an expression that can be used like any other expression. You can use a function call wherever it is legal to use an expression of the type specified for the value

function call

returned by the function. For example, the value returned by `sqrt` is of type *double*. Thus, the following is legal (although perhaps stingy):

```
bonus = sqrt(sales)/10;
```

`sales` and `bonus` are variables that would normally be of type *double*. The function call `sqrt(sales)` is a single item, just as if it were enclosed in parentheses. Thus, the above assignment statement is equivalent to:

```
bonus = (sqrt(sales))/10;
```

You can also use a function call directly in a `cout` statement, as in the following:

```
cout << "The side of a square with area " << area
     << " is " << sqrt(area);
```

Display 3.1 contains a complete program that uses the predefined function `sqrt`. The program computes the size of the largest square dog house that can be built for the amount of money the user is willing to spend. The program asks the user for an amount of money, and then determines how many square feet of floor space can be purchased for that amount of money. That calculation yields an area in square feet for the floor area of the dog house. The function `sqrt` yields the length of one side of the dog house floor.

Notice that there is another new element in the program in Display 3.1:

```
#include <cmath>
```

Function Call

A function call is an expression consisting of the function name followed by arguments enclosed in parentheses. If there is more than one argument, the arguments are separated by commas. A function call is an expression that can be used like any other expression of the type specified for the value returned by the function.

Syntax

Function_Name(*Argument_List*)

where the *Argument_List* is a comma-separated list of arguments:

Argument_1, *Argument_2*, . . . , *Argument_Last*

Examples

```
side = sqrt(area);
cout << "2.5 to the power 3.0 is "
     << pow(2.5, 3.0);
```

Display 3.1 A Function Call


```
//Computes the size of a dog house that can be purchased
//given the user's budget.
#include <iostream>
#include <cmath>
using namespace std;

int main( )
{
    const double COST_PER_SQ_FT = 10.50;
    double budget, area, length_side;

    cout << "Enter the amount budgeted for your dog house $";
    cin >> budget;

    area = budget/COST_PER_SQ_FT;
    length_side = sqrt(area);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For a price of $" << budget << endl
         << "I can build you a luxurious square dog house\n"
         << "that is " << length_side
         << " feet on each side.\n";

    return 0;
}
```

Sample Dialogue

```
Enter the amount budgeted for your dog house $25.00
For a price of $25.00
I can build you a luxurious square dog house
that is 1.54 feet on each side.
```

That line looks very much like the line

```
#include <iostream>
```

#include directive and header file

and, in fact, these two lines are the same sort of thing. As we noted in Chapter 2, such lines are called `include` **directives.** The name inside the angular brackets < > is the name of a file known as a **header file.** A header file for a library provides the compiler with certain basic information about the library, and an `include` directive delivers this information to the compiler. This enables the linker to find object code for the functions in the library so that it can correctly link the library to your program. For example, the library iostream contains the definitions of `cin` and `cout`, and the header file for the iostream library is called `iostream`. The math library contains the definition of the function `sqrt` and a number of other mathematical functions, and the header file for this library is `cmath`. If your program uses a predefined function from some library, then it must contain a directive that names the header file for that library, such as the following:

```
#include <cmath>
```

Be sure to follow the syntax illustrated in our examples. Do not forget the symbols < and >; they are the same symbols as the less-than and greater-than symbols. There should be no space between the < and the filename, nor between the filename and the >. Also, some compilers require that directives have no spaces around the #, so it is always safest to place the # at the very start of the line and not to put any space between the # and the word `include`. These `#include` directives are normally placed at the beginning of the file containing your program.

As we noted before, the directive

```
#include <iostream>
```

requires that you also use the following *`using`* directive:

```
using namespace std;
```

This is because the definitions of names like `cin` and `cout`, which are given in `iostream`, define those names to be part of the `std` namespace. This is true of most standard libraries. If you have an `include` directive for a standard library such as

```
#include <cmath>
```

then you probably need the *`using`* directive:

```
using namespace std;
```

There is no need to use multiple copes of this *`using`* directive when you have multiple include directives.

#include may not be enough

Usually, all you need to do to use a library is to place an `include` directive and a *`using`* directive for that library in the file with your program. If things work with

just the `include` directive and the *`using`* directive, you need not worry about doing anything else. However, for some libraries on some systems you may need to give additional instructions to the compiler or to explicitly run a linker program to link in the library. Early C and C++ compilers did not automatically search all libraries for linking. The details vary from one system to another, so you will have to check your manual or a local expert to see exactly what is necessary.

Some people will tell you that `include` directives are not processed by the compiler but are processed by a **preprocessor.** They're right, but the difference is more of a word game than anything that need concern you. On almost all compilers the preprocessor is called automatically when you compile your program.

preprocessor

A few predefined functions are described in Display 3.2. More predefined functions are described in Appendix 4. Notice that the absolute value functions `abs` and `labs` are in the library with header file `cstdlib`, so any program that uses either of these functions must contain the following directive:

`abs` and `labs`

```
#include <cstdlib>
```

All the other functions listed are in the library with header file `cmath`, just like `sqrt`.

Also notice that there are three absolute value functions. If you want to produce the absolute value of a number of type *`int`*, you use `abs`; if you want to produce the absolute value of a number of type *`long`*, you use `labs`; and if you want to produce the absolute value of a number of type *`double`*, you use `fabs`. To complicate things even more, `abs` and `labs` are in the library with header file `cstdlib`, while `fabs` is in the library with header file `cmath`. `fabs` is an abbreviation for *floating-point absolute value*. Recall that numbers with a fraction after the decimal point, such as numbers of type *`double`*, are often called *floating-point numbers*.

`fabs`

Another example of a predefined function is `pow`, which is in the library with header file `cmath`. The function `pow` can be used to do exponentiation in C++. For example, if you want to set a variable `result` equal to x^y, you can use the following:

`pow`

```
result = pow(x, y);
```

Hence, the following three lines of program code will output the number `9.0` to the screen, because $(3.0)^{2.0}$ is `9.0`:

```
double result, x = 3.0, y = 2.0;
result = pow(x, y);
cout << result;
```

Notice that the above call to `pow` returns `9.0`, not `9`. The function `pow` always returns a value of type *`double`*, not of type *`int`*. Also notice that the function `pow` requires two arguments. A function can have any number of arguments. Moreover, every argument position has a specified type and the argument used in a function call

arguments have a type

Display 3.2 Some Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
`sqrt`	square root	*double*	*double*	`sqrt(4.0)`	`2.0`	`cmath`
`pow`	powers	*double*	*double*	`pow(2.0,3.0)`	`8.0`	`cmath`
`abs`	absolute value for *int*	*int*	*int*	`abs(-7)` `abs(7)`	`7` `7`	`cstdlib`
`labs`	absolute value for *long*	*long*	*long*	`labs(-70000)` `labs(70000)`	`70000` `70000`	`cstdlib`
`fabs`	absolute value for *double*	*double*	*double*	`fabs(-7.5)` `fabs(7.5)`	`7.5` `7.5`	`cmath`
`ceil`	ceiling (round up)	*double*	*double*	`ceil(3.2)` `ceil(3.9)`	`4.0` `4.0`	`cmath`
`floor`	floor (round down)	*double*	*double*	`floor(3.2)` `floor(3.9)`	`3.0` `3.0`	`cmath`

should be of that type. In many cases, if you use an argument of the wrong type, then some automatic type conversion will be done for you by C++. However, the results may not be what you intended. When you call a function, you should use arguments of the type specified for that function. One exception to this caution is the automatic conversion of arguments from type *int* to type *double*. In many situations, including calls to the function pow, you can safely use an argument of type *int* when an argument of type *double* is specified.

restrictions on pow

Many implementations of pow have a restriction on what arguments can be used. In these implementations, if the first argument to pow is negative, then the second argument must be a whole number. Since you probably have enough other things to worry about when learning to program, it might be easiest and safest to use pow only when the first argument is nonnegative.

Type Casting

Division may require the type *double*

Recall that 9/2 is integer division, and evaluates to 4, not 4.5. If you want division to produce an answer of type *double* (that is, including the fractional part after the decimal point), then at least one of the two numbers in the division must be of type *double*. For example, 9/2.0 evaluates to 4.5. If one of the two numbers is given as a constant, you can simply add a decimal point and a zero to one (or both) numbers, and the division will then produce a value that includes the digits after the decimal point.

But what if both of the operands in a division are variables, as in the following?

```
int total_candy, number_of_people;
double candy_per_person;
 <The program somehow sets the value of total_candy to 9
     and the value of number_of_people to 2.
     It does not matter how the program does this.>
candy_per_person = total_candy/number_of_people;
```

Unless you convert the value in one of the variables total_candy or number_of_people to a value of type *double*, then the result of the division will be 4, not 4.5 as it should be. The fact that the variable candy_per_person is of type *double* does not help. The value of 4 obtained by division will be converted to a value of type *double* before it is stored in the variable candy_per_person, but that will be too late. The 4 will be converted to 4.0 and the final value of candy_per_person will be 4.0, not 4.5. If one of the quantities in the division were a constant, you could add a decimal point and a zero to convert the constant to type *double*, but in this case both quantities are variables. Fortunately, there is a way to convert from type *int* to type *double* that you can use with either a constant or a variable.

In C++ you can tell the computer to convert a value of type *int* to a value of type *double*. The way that you write "Convert the value 9 to a value of type *double*" is

```
static_cast<double>(9)
```

The notation *static_cast<double>* is a kind of predefined function that converts a value of some other type, such as 9, to a value of type *double*, in this case 9.0. An expression such as *static_cast<double>*(9) is called a **type cast.** You can use a variable or other expression in place of the 9. You can use other type names besides *double* to obtain a type cast to some type other than *double*, but we will postpone that topic until later.

type casting

For example, in the following we use a type cast to change the type of 9 from *int* to *double* and so the value of answer is set to 4.5:

```
double answer;
answer = static_cast<double>(9)/2;
```

Type casting applied to a constant, such as 9, can make your code easier to read, since it makes your intended meaning clearer. But type casting applied to constants of type *int* does not give you any additional power. You can use `9.0` instead of `static_cast<double>(9)` when you want to convert 9 to a value of type *double*. However, if the division involves only variables, then type casting may be your only sensible alternative. Using type casting, we can rewrite our earlier example so that the variable `candy_per_person` receives the correct value of `4.5`, instead of `4.0`; in order to do this, the only change we need is the replacement of `total_candy` with `static_cast<double>(total_candy)`, as shown in what follows:

```
int total_candy, number_of_people;
double candy_per_person;
 <The program somehow sets the value of total_candy to 9
    and the value of number_of_people to 2.
    It does not matter how the program does this.>
candy_per_person =
            static_cast<double>(total_candy)/number_of_people;
```

Warning!

Notice the placement of parentheses in the type casting used in the above code. You want to do the type casting before the division so that the division operator is working on a value of type *double*. If you wait until after the division is completed, then the digits after the decimal point are already lost. If you mistakenly use the following for the last line of the above code, then the value of `candy_per_person` will be `4.0`, not `4.5`.

```
candy_per_person =
static_cast<double>(total_candy/number_of_people); //WRONG!
```

Older Form of Type Casting

double used as a function

The use of `static_cast<double>`, as we discussed in the previous section, is the preferred way to perform a type cast. However, older versions of C++ used a different notation for type casting. This older notation simply uses the type name as if it were a function name, so `double(9)` returns `9.0`. Thus, if `candy_per_person` is a variable of type *double*, and if both `total_candy` and `number_of_people` are variables of type *int*, then the following two assignment statements are equivalent:

```
candy_per_person =
            static_cast<double>(total_candy)/number_of_people;
```

and

```
candy_per_person =
            double(total_candy)/number_of_people;
```

A Function to Convert from *int* to *double*

The notation *static_cast<double>* can be used as a predefined function and will convert a value of some other type to a value of type *double*. For example, *static_cast<double>*(2) returns 2.0. This is called **type casting.** (Type casting can be done with types other than *double*, but until later in this book, we will only do type casting with the type *double*.)

Syntax

```
static_cast<double>(Expression_of_Type_int)
```

Example

```
int total_pot, number_of_winners;
double your_winnings;
  . . .
your_winnings =
      static_cast<double>(total_pot)/number_of_winners;
```

Although *static_cast<double>*(total_candy) and *double*(total_candy) are more or less equivalent, you should use the *static_cast<double>* form, since the form *double*(total_candy) may be discontinued in later versions of C++.

PITFALL Integer Division Drops the Fractional Part

In integer division, such as computing 11/2, it is easy to forget that 11/2 gives 5, not 5.5. The result is the next lower integer, regardless of the subsequent use of this result. For example,

```
double d;
d = 11/2;
```

Here, the division is done using integer divide; the result of the division is 5, which is converted to *double*, then assigned to d. The fractional part is not generated. Observe that the fact that d is of type *double* does not change the division result. The variable d receives the value 5.0, not 5.5.

SELF-TEST EXERCISES

1 Determine the value of each of the following arithmetic expressions:

`sqrt(16.0)`	`sqrt(16)`	`pow(2.0, 3.0)`
`pow(2, 3)`	`pow(2.0, 3)`	`pow(1.1, 2)`
`abs(3)`	`abs(–3)`	`abs(0)`
`fabs(–3.0)`	`fabs(–3.5)`	`fabs(3.5)`
`ceil(5.1)`	`ceil(5.8)`	`floor(5.1)`
`floor(5.8)`	`pow(3.0, 2)/2.0`	`pow(3.0, 2)/2`
`7/abs(–2)`	`(7 + sqrt(4.0))/3.0`	`sqrt(pow(3, 2))`

2 Convert each of the following mathematical expressions to a C++ arithmetic expression:

$\sqrt{x + y}$ x^{y+7} $\sqrt{area + fudge}$

$\frac{\sqrt{time + tide}}{nobody}$ $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$ $|x - y|$

3 Write a complete C++ program to compute and output the square root of `PI`; `PI` is approximately 3.14159. The `const double PI` is predefined in `cmath`. You are encouraged to use this predefined constant.

4 Write and compile short programs to test the following issues:

a. Determine if your compiler will allow the `#include <iostream>` anywhere on the line, or if the `#` needs to be flush with the left margin.

b. Determine whether your compiler will allow space between the `#` and the `include`.

3.3 Programmer-Defined Functions

A custom tailored suit always fits better than one off the rack.

MY UNCLE, THE TAILOR

In the previous section we told you how to use predefined functions. In this section we tell you how to define your own functions.

Function Definitions

You can define your own functions, either in the same file as the `main` part of your program or in a separate file so that the functions can be used by several different programs. The definition is the same in either case, but for now, we will assume that the function definition will be in the same file as the `main` part of your program.

Display 3.3 contains a sample function definition in a complete program that demonstrates a call to the function. The function is called `total_cost`. The function takes two arguments—the price for one item and number of items for a purchase. The function returns the total cost, including sales tax, for that many items at the specified price. The function is called in the same way a predefined function is called. The description of the function, which the programmer must write, is a bit more complicated.

The description of the function is given in two parts that are called the *function declaration* and the *function definition*. The **function declaration** (also known as the **function prototype**) describes how the function is called. C++ requires that either the complete function definition or the function declaration appears in the code before the function is called. The function declaration for the function `total_cost` is in color at the top of Display 3.3 and is reproduced below:

function declaration

```
double total_cost(int number_par, double price_par);
```

The function declaration tells you everything you need to know in order to write a call to the function. It tells you the name of the function, in this case `total_cost`. It tells you how many arguments the function needs and what type the arguments should be; in this case, the function `total_cost` takes two arguments, the first one of type `int` and the second one of type `double`. The identifiers `number_par` and `price_par` are called *formal parameters*. A **formal parameter** is used as a kind of blank, or placeholder, to stand in for the argument. When you write a function declaration you do not know what the arguments will be, so you use the formal parameters in place of the arguments. The names of the formal parameters can be any valid identifiers, but for a while we will end our formal parameter names with `_par` so that it will be easier for us to distinguish them from other items in a program. Notice that a function declaration ends with a semicolon.

formal parameter

The first word in a function declaration specifies the type of the value returned by the function. Thus, for the function `total_cost`, the type of the value returned is `double`.

type for value returned

As you can see, the function call in Display 3.3 satisfies all the requirements given by its function declaration. Let's take a look. The function call is in the following line:

```
bill = total_cost(number, price);
```

Display 3.3 A Function Definition (*part 1 of 2*)


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

double total_cost(int number_par, double price_par);
//Computes the total cost, including 5% sales tax,
//on number_par items at a cost of price_par each.

int main( )
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = total_cost(number, price);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}

double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price_par * number_par;
    return (subtotal + subtotal*TAX_RATE);
}
```

Display 3.3 A Function Definition (*part 2 of 2*)

Sample Dialogue

```
Enter the number of items purchased: 2
Enter the price per item: $10.10
2 items at  $10.10 each.
Final bill, including tax, is $21.21
```

The function call is the expression on the right-hand side of the equal sign. The function name is `total_cost`, and there are two arguments: The first argument is of type `int`, the second argument is of type `double`, and since the variable `bill` is of type `double`, it looks like the function returns a value of type `double` (which it does). All that detail is determined by the function declaration.

function declaration comment

The compiler does not care whether there's a comment along with the function declaration, but you should always include a comment that explains what value is returned by the function.

Function Declaration

A **function declaration** tells you all you need to know to write a call to the function. A function declaration is required to appear in your code prior to a call to a function whose definition has not yet appeared. Function declarations are normally placed before the `main` part of your program.

Syntax

Type_Returned Function_Name`(`*Parameter_List*`);` ← *Do not forget this semicolon.*
Function_Declaration_Comment

where the *Parameter_List* is a comma-separated list of parameters:

Type_1 Formal_Parameter_1`,` *Type_2 Formal_Parameter_2*`,...`

`...,` *Type_Last Formal_Parameter_Last*

Example

```
double total_weight(int number, double weight_of_one);
//Returns the total weight of number items that
//each weigh weight_of_one.
```

function definition

In Display 3.3 the function definition is in color at the bottom of the display. A **function definition** describes how the function computes the value it returns. If you think of a function as a small program within your program, then the function definition is like the code for this small program. In fact, the syntax for the definition of a function is very much like the syntax for the `main` part of a program. A function definition consists of a *function header* followed by a *function body*. The **function header** is written the same way as the function declaration, except that the header does *not* have a semicolon at the end. This makes the header a bit repetitious, but that's OK.

function header

function body

Although the function declaration tells you all you need to know to write a function call, it does not tell you what value will be returned. The value returned is determined by the statements in the *function body*. The **function body** follows the function header and completes the function definition. The function body consists of declarations and executable statements enclosed within a pair of braces. Thus, the function body is just like the body of the `main` part of a program. When the function is called, the argument values are plugged in for the formal parameters and then the statements in the body are executed. The value returned by the function is determined when the function executes a *`return` statement*. (The details of this "plugging in" will be discussed in a later section.)

`return` statement

A ***`return`* statement** consists of the keyword *`return`* followed by an expression. The function definition in Display 3.3 contains the following *`return`* statement:

```
return (subtotal + subtotal*TAX_RATE);
```

When this *`return`* statement is executed, the value of the following expression is returned as the value of the function call:

```
(subtotal + subtotal*TAX_RATE)
```

The parentheses are not needed. The program will run exactly the same if the *`return`* statement is written as follows:

```
return subtotal + subtotal*TAX_RATE;
```

However, on larger expressions, the parentheses make the *`return`* statement easier to read. For consistency, some programmers advocate using these parentheses even on simple expressions. In the function definition in Display 3.3 there are no statements after the *`return`* statement, but if there were, they would not be executed. When a *`return`* statement is executed, the function call ends.

anatomy of a function call

Let's see exactly what happens when the following function call is executed in the program shown in Display 3.3:

```
bill = total_cost(number, price);
```

First, the values of the arguments `number` and `price` are plugged in for the formal parameters; that is, the values of the arguments `number` and `price` are substituted in for `number_par` and `price_par`. In the Sample Dialogue, `number` receives the value `2` and `price` receives the value `10.10`. So `2` and `10.10` are substituted for `number_par` and `price_par`, respectively. This substitution process is known as the **call-by-value mechanism,** and the formal parameters are often referred to as **call-by-value formal parameters,** or simply as **call-by-value parameters.** There are three things that you should note about this substitution process:

call-by-value

1. It is the values of the arguments that are plugged in for the formal parameters. If the arguments are variables, the values of the variables, not the variables themselves, are plugged in.
2. The first argument is plugged in for the first formal parameter in the parameter list, the second argument is plugged in for the second formal parameter in the list, and so forth.
3. When an argument is plugged in for a formal parameter (for instance when `2` is plugged in for `number_par`), the argument is plugged in for *all* instances of the formal parameter that occur in the function body (for instance `2` is plugged in for `number_par` each time it appears in the function body).

The entire process involved in the function call shown in Display 3.3 is described in detail in Display 3.4.

Alternate Form for Function Declarations

You are not required to list formal parameter names in a function declaration. The following two function declarations are equivalent:

```
double total_cost(int number_par, double price_par);
```

and the equivalent

```
double total_cost(int, double);
```

We will always use the first form so that we can refer to the formal parameters in the comment that accompanies the function declaration. However, you will often see the second form in manuals that describe functions.[1]

This alternate form applies only to function declarations. *Function headers must always list the formal parameter names.*

[1] All C++ needs to be able to enable your program to link to the library or your function is the function name and sequence of types of the formal parameters. The formal parameter names are important only to the function definition. However, programs should communicate to programmers as well as to compilers. It is frequently very helpful in understanding a function to use the name that the programmer attaches to the function's data.

Display 3.4 Details of a Function Call (*part 1 of 2*)

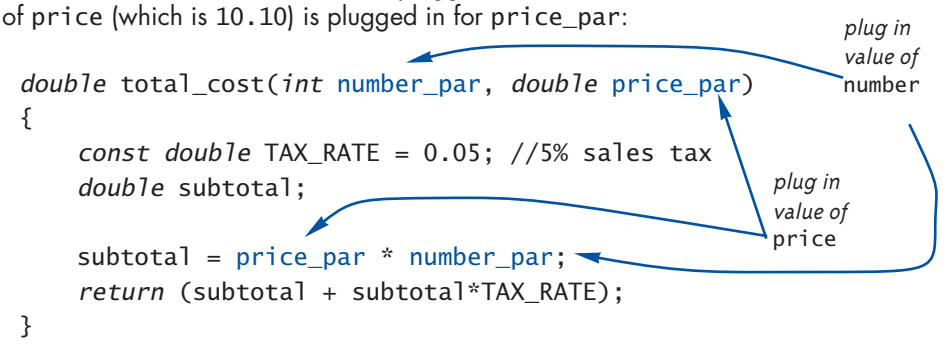
Anatomy of the Function Call in Display 3.3

0 Before the function is called, the values of the variables `number` and `price` are set to 2 and 10.10, by `cin` statements (as you can see in the Sample Dialogue in Display 3.3).

1 The following statement, which includes a function call, begins executing:

```
bill = total_cost(number, price);
```

2 The value of `number` (which is 2) is plugged in for `number_par` and the value of `price` (which is 10.10) is plugged in for `price_par`:




```
double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price_par * number_par;
    return (subtotal + subtotal*TAX_RATE);
}
```

producing the following:

```
double total_cost(int 2, double 10.10)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = 10.10 * 2;
    return (subtotal + subtotal*TAX_RATE);
}
```

Display 3.4 Details of a Function Call (*part 2 of 2*)

Anatomy of the Function Call in Display 3.3 (*concluded*)

3 The body of the function is executed, that is, the following is executed:

```
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = 10.10 * 2;
    return (subtotal + subtotal*TAX_RATE);
}
```

4 When the *return* statement is executed, the value of the expression after *return* is the value returned by the function. In this case when

```
return (subtotal + subtotal*TAX_RATE);
```

is executed, the value of `(subtotal + subtotal*TAX_RATE)`, which is `21.21`, is returned by the function call

```
total_cost(number, price)
```

and so the value of `bill` (on the left-hand side of the equal sign) is set equal to `21.21` when the following statement finally ends:

```
bill = total_cost(number, price);
```

A Function Is Like a Small Program

To understand functions, keep the following three points in mind:

- A function definition is like a small program and calling the function is the same thing as running this "small program."
- A function uses formal parameters, rather than `cin`, for input. The arguments to the function are the input and they are plugged in for the formal parameters.
- A function (of the kind discussed in this chapter) does not normally send any output to the screen, but it does send a kind of "output" back to the program. The function returns a value, which is like the "output" for the function. The function uses a *return* statement instead of a `cout` statement for this "output."

PITFALL Arguments in the Wrong Order

When a function is called, the computer substitutes the first argument for the first formal parameter, the second argument for the second formal parameter, and so forth. It does not check for reasonableness. If you confuse the order of the arguments in a function call, the program will not do what you want it to do. In order to see what can go wrong, consider the program in Display 3.5. The programmer who wrote that program carelessly reversed the order of the arguments in the call to the function `grade`. The function call should have been

```
letter_grade = grade(score, need_to_pass);
```

This is the only mistake in the program. Yet, some poor student has been mistakenly failed in a course because of this careless mistake. The function `grade` is so simple that you might expect this mistake to be discovered by the programmer when the program is tested. However, if `grade` were a more complicated function, the mistake might easily go unnoticed.

If the type of an argument does not match the formal parameter, then the compiler may give you a warning message. Unfortunately, not all compilers will give such warning messages. Moreover, in a situation like the one in Display 3.5, no compiler will complain about the ordering of the arguments, because the function argument types will match the formal parameter types no matter what order the arguments are in.

Function Definition-Syntax Summary

Function declarations are normally placed before the `main` part of your program and function definitions are normally placed after the `main` part of your program (or, as we will see later in this book, in a separate file). Display 3.6 gives a summary of the syntax for a function declaration and definition. There is actually a bit more freedom than that display indicates. The declarations and executable statements in the function definition can be intermixed, as long as each variable is declared before it is used. The rules about intermixing declarations and executable statements in a function definition are the same as they are for the `main` part of a program. However, unless you have reason to do otherwise, it is best to place the declarations first, as indicated in Display 3.6.

return statement

Since a function does not return a value until it executes a *return* statement, a function must contain one or more *return* statements in the body of the function. A function definition may contain more than one *return* statement. For example, the body of the code might contain an *if-else* statement, and each branch of the *if-else* statement might contain a different *return* statement, as illustrated in Display 3.5.

Display 3.5 Incorrectly Ordered Arguments (*part 1 of 2*)

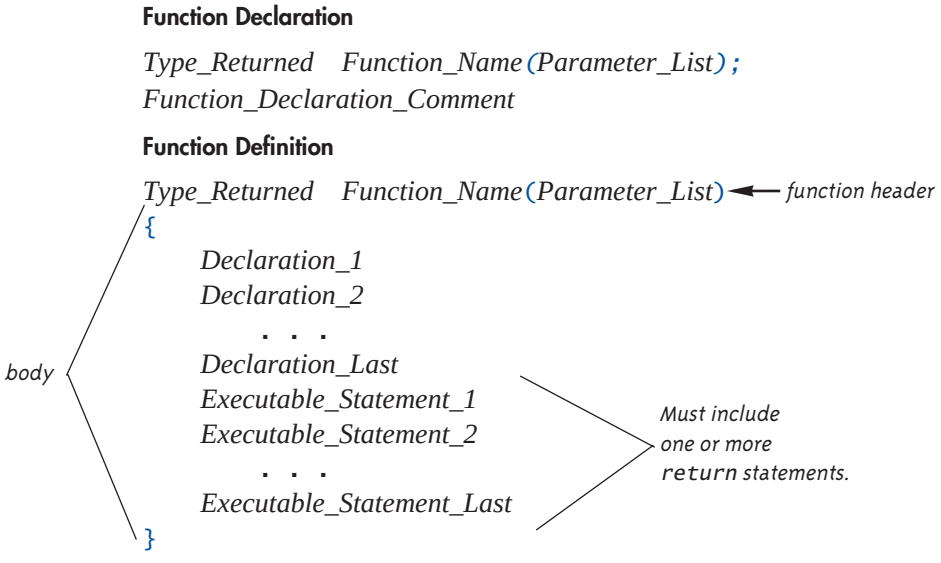

```
//Determines user's grade. Grades are Pass or Fail.
#include <iostream>
using namespace std;

char grade(int received_par, int min_score_par);
//Returns 'P' for passing, if received_par is
//min_score_par or higher. Otherwise returns 'F' for failing.

int main()
{
    int score, need_to_pass;
    char letter_grade;

    cout << "Enter your score"
         << " and the minimum needed to pass:\n";
    cin >> score >> need_to_pass;

    letter_grade = grade(need_to_pass, score);

    cout << "You received a score of " << score << endl
         << "Minimum to pass is " << need_to_pass << endl;

    if (letter_grade == 'P')
        cout << "You Passed. Congratulations!\n";
    else
        cout << "Sorry. You failed.\n";

    cout << letter_grade
         << " will be entered in your record.\n";

    return 0;
}

char grade(int received_par, int min_score_par)
{
    if (received_par >= min_score_par)
        return 'P';
    else
        return 'F';
}
```

Display 3.5 Incorrectly Ordered Arguments (*part 2 of 2*)

Sample Dialogue

```
Enter your score and the minimum needed to pass:
98 60
You received a score of 98
Minimum to pass is 60
Sorry. You failed.
F will be entered in your record.
```

Display 3.6 Syntax for a Function That Returns a Value



spacing and line breaks

Any reasonable pattern of spaces and line breaks in a function definition will be accepted by the compiler. However, you should use the same rules for indenting and laying out a function definition as you use for the `main` part of a program. In particular, notice the placement of braces `{}` in our function definitions and in Display 3.6.

The opening and closing braces that mark the ends of the function body are each placed on a line by themselves. This sets off the function body.

More about Placement of Function Definitions

We have discussed where function definitions and function declarations are normally placed. Under normal circumstances these are the best locations for the function declarations and function definitions. However, the compiler will accept programs with the function definitions and function declarations in certain other locations. A more precise statement of the rules is as follows: Each function call must be preceded by either a function declaration for that function or the definition of the function. For example, if you place all of your function definitions before the `main` part of the program, then you need not include any function declarations. Knowing this more general rule will help you to understand C++ programs you see in some other books, but you should follow the example of the programs in this book. The style we are using sets the stage for learning how to build your own libraries of functions, which is the style that most C++ programmers use.

SELF-TEST EXERCISES

5 What is the output produced by the following program?

```
#include <iostream>
using namespace std;
char mystery(int first_par, int second_par);
int main( )
{
    cout << mystery(10, 9) << "ow\n";
    return 0;
}

char mystery(int first_par, int second_par)
{
    if (first_par >= second_par)
        return 'W';
    else
        return 'H';
}
```

6 Write a function declaration and a function definition for a function that takes three arguments, all of type *int*, and that returns the sum of its three arguments.

7 Write a function declaration and a function definition for a function that takes one argument of type `int` and one argument of type `double`, and that returns a value of type `double` that is the average of the two arguments.

8 Write a function declaration and a function definition for a function that takes one argument of type `double`. The function returns the character value `'P'` if its argument is positive and returns `'N'` if its argument is zero or negative.

9 Carefully describe the call-by-value parameter mechanism.

10 List the similarities and differences between use of a predefined (that is, library) function and a user-defined function.

3.4 Procedural Abstraction

The cause is hidden, but the result is well known.

OVID, *METAMORPHOSES IV*

The Black Box Analogy

A person who uses a program should not need to know the details of how the program is coded. Imagine how miserable your life would be if you had to know and remember the code for the compiler you use. A program has a job to do, such as compile your program or check the spelling of words in your paper. You need to know *what* the program's job is so that you can use the program, but you do not (or at least should not) need to know *how* the program does its job. A function is like a small program and should be used in a similar way. A programmer who uses a function in a program needs to know *what* the function does (such as calculate a square root or convert a temperature from degrees Fahrenheit to degrees Celsius), but should not need to know *how* the function accomplishes its task. This is often referred to as treating the function like a *black box*.

black box

Calling something a **black box** is a figure of speech intended to convey the image of a physical device that you know how to use but whose method of operation is a mystery, because it is enclosed in a black box and you cannot see inside the box (and cannot pry it open!). If a function is well designed, the programmer can use the function as if it were a black box. All the programmer needs to know is that if he or she puts appropriate arguments into the black box, then an appropriate returned value will come out of the black box. Designing a function so that it can be used as a black box is sometimes called **information hiding** to emphasize the fact that the programmer acts as if the body of the function were hidden from view.

information hiding

Display 3.7 contains the function declaration and two different definitions for a function named new_balance. As the function declaration comment explains, the function new_balance calculates the new balance in a bank account when simple interest is added. For instance, if an account starts with $100, and 4.5% interest is posted to the account, then the new balance is $104.50. Hence, the following code will change the value of vacation_fund from 100.00 to 104.50:

```
vacation_fund = 100.00;
vacation_fund = new_balance(vacation_fund, 4.5);
```

Display 3.7 Definitions That Are Black-Box Equivalent

Function Declaration

```
double new_balance(double balance_par, double rate_par);
//Returns the balance in a bank account after
//posting simple interest. The formal parameter balance_par is
//the old balance. The formal parameter rate_par is the interest rate.
//For example, if rate_par is 5.0, then the interest rate is 5%
//and so new_balance(100, 5.0) returns 105.00.
```

Definition 1

```
double new_balance(double balance_par, double rate_par)
{
    double interest_fraction, interest;

    interest_fraction = rate_par/100;
    interest = interest_fraction*balance_par;
    return (balance_par + interest);
}
```

Definition 2

```
double new_balance(double balance_par, double rate_par)
{
    double interest_fraction, updated_balance;

    interest_fraction = rate_par/100;
    updated_balance = balance_par*(1 + interest_fraction);
    return updated_balance;
}
```

It does not matter which of the implementations of `new_balance` shown in Display 3.7 that a programmer uses. The two definitions produce functions that return exactly the same values. We may as well place a black box over the body of the function definition so that the programmer does not know which implementation is being used. In order to use the function `new_balance`, all the programmer needs to read is the function declaration and the accompanying comment.

procedural abstraction

Writing and using functions as if they were black boxes is also called **procedural abstraction.** When programming in C++ it might make more sense to call it *functional abstraction*. However, *procedure* is a more general term than *function*. Computer scientists use the term *procedure* for all "function-like" sets of instructions, and so they use the term *procedural abstraction*. The term *abstraction* is intended to convey the idea that, when you use a function as a black box, you are abstracting away the details of the code contained in the function body. You can call this technique *the black box principle* or *the principle of procedural abstraction* or *information hiding*. The three terms mean the same thing. Whatever you call this principle, the important point is that you should use it when designing and writing your function definitions.

Procedural Abstraction

When applied to a function definition, the principle of **procedural abstraction** means that your function should be written so that it can be used like a **black box.** This means that the programmer who uses the function should not need to look at the body of the function definition to see how the function works. The function declaration and the accompanying comment should be all the programmer needs to know in order to use the function. To ensure that your function definitions have this important property, you should strictly adhere to the following rules:

How to Write a Black-Box Function Definition (That Returns a Value)

- The function declaration comment should tell the programmer any and all conditions that are required of the arguments to the function and should describe the value that is returned by the function when called with these arguments.
- All variables used in the function body should be declared in the function body. (The formal parameters do not need to be declared, because they are listed in the function declaration.)

Programming TIP
Choosing Formal Parameter Names

The principle of procedural abstraction says that functions should be self-contained modules that are designed separately from the rest of the program. On large programming projects a different programmer may be assigned to write each function. The programmer should choose the most meaningful names he or she can find for formal parameters. The arguments that will be substituted for the formal parameters may well be variables in the `main` part of the program. These variables should also be given meaningful names, often chosen by someone other than the programmer who writes the function definition. This makes it likely that some or all arguments will have the same names as some of the formal parameters. This is perfectly acceptable. No matter what names are chosen for the variables that will be used as arguments, these names will not produce any confusion with the names used for formal parameters. After all, the functions will use only the values of the arguments. When you use a variable as a function argument, the function takes only the value of the variable and disregards the variable name.

Now that you know you have complete freedom in choosing formal parameter names, we will stop placing a "`_par`" at the end of each formal parameter name. For example, in Display 3.8 we have rewritten the definition for the function `total_cost` from Display 3.3 so that the formal parameters are named `number` and `price` rather than `number_par` and `price_par`. If you replace the function declaration and definition of the function `total_cost` that appear in Display 3.3 with the versions in Display 3.8, then the program will perform in exactly the same way, even though there will be formal parameters named `number` and `price` and there will be variables in the `main` part of the program that are also named `number` and `price`.

CASE STUDY Buying Pizza

The large "economy" size of an item is not always a better buy than the smaller size. This is particularly true when buying pizzas. Pizza sizes are given as the diameter of the pizza in inches. However, the quantity of pizza is determined by the area of the pizza and the area is not proportional to the diameter. Most people cannot easily estimate the difference in area between a ten-inch pizza and a twelve-inch pizza, and so cannot easily determine which size is the best buy—that is, which size has the lowest price per square inch. In this case study we will design a program that compares two sizes of pizza to determine which is the better buy.

Display 3.8 Simpler Formal Parameter Names

Function Declaration

```
double total_cost(int number, double price);
//Computes the total cost, including 5% sales tax, on
//number items at a cost of price each.
```

Function Definition

```
double total_cost(int number, double price)
{
    const double TAX_RATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = price * number;
    return (subtotal + subtotal*TAX_RATE);
}
```

PROBLEM DEFINITION

The precise specification of the program input and output are as follows:

INPUT

The input will consist of the diameter in inches and the price for each of two sizes of pizza.

OUTPUT

The output will give the cost per square inch for each of the two sizes of pizza and will tell which is the better buy, that is, which has the lowest cost per square inch. (If they are the same cost per square inch, we will consider the smaller one to be the better buy.)

ANALYSIS OF THE PROBLEM

We will use top-down design to divide the task to be solved by our program into the following subtasks:

Subtask 1: Get the input data for both the small and large pizzas.
Subtask 2: Compute the price per square inch for the small pizza.
Subtask 3: Compute the price per square inch for the large pizza.
Subtask 4: Determine which is the better buy.
Subtask 5: Output the results.

subtasks 2 and 3

Notice subtasks 2 and 3. They have two important properties:

i. They are exactly the same task. The only difference is that they use different data to do the computation. The only things that change between subtask 2 and subtask 3 are the size of the pizza and its price.

ii. The result of subtask 2 and the result of subtask 3 are each a single value, the price per square inch of the pizza.

when to define a function

Whenever a subtask takes some values, such as some numbers, and returns a single value, it is natural to implement the subtask as a function. Whenever two or more such subtasks perform the same computation, they can be implemented as the same function called with different arguments each time it is used. We therefore decide to use a function called `unitprice` to compute the price per square inch of a pizza. The function declaration and explanatory comment for this function will be as follows:

```
double unitprice(int diameter, double price);
//Returns the price per square inch of a pizza. The formal
//parameter named diameter is the diameter of the pizza in
//inches. The formal parameter named price is the price of
//the pizza.
```

ALGORITHM DESIGN

subtask 1

Subtask 1 is straightforward. The program will simply ask for the input values and store them in four variables, which we will call `diameter_small`, `diameter_large`, `price_small`, and `price_large`.

subtasks 4 and 5

Subtask 4 is routine. To determine which pizza is the best buy, we just compare the cost per square inch of the two pizzas using the less-than operator. Subtask 5 is a routine output of the results.

subtasks 2 and 3

Subtasks 2 and 3 are implemented as calls to the function `unitprice`. Next, we design the algorithm for this function. The hard part of the algorithm is determining the area of the pizza. Once we know the area, we can easily determine the price per square inch using division, as follows:

```
price/area
```

where `area` is a variable that holds the area of the pizza. The above expression will be the value returned by the function `unitprice`. But we still need to formulate a method for computing the area of the pizza.

A pizza is basically a circle (made up of bread, cheese, sauce, and so forth). The area of a circle (and hence of a pizza) is πr^2, where r is the radius of the circle, and π is the number called "pi," which is approximately equal to 3.14159. The radius is one half of the diameter.

The algorithm for the function `unitprice` can be outlined as follows:

Algorithm Outline for the Function `unitprice`

1. Compute the radius of the pizza.
2. Compute the area of the pizza using the formula πr^2.
3. Return the value of the expression `(price/area)`.

We will give this outline a bit more detail before translating it into C++ code. We will express this more detailed version of our algorithm in *pseudocode*.

pseudocode

Pseudocode is a mixture of C++ and ordinary English. Pseudocode allows us to make our algorithm precise without worrying about the details of C++ syntax. We can then easily translate our pseudocode into C++ code. In our pseudocode, `radius` and `area` will be variables for holding the values indicated by their names.

Pseudocode for the Function `unitprice`

```
radius = one half of diameter;
area = π * radius * radius;
return (price/area);
```

That completes our algorithm for `unitprice`. We are now ready to convert our solutions to subtasks 1 through 5 into a complete C++ program.

CODING

Coding subtask 1 is routine, so we next consider subtasks 2 and 3. Our program can implement subtasks 2 and 3 by the following two calls to the function `unitprice`:

```
unitprice_small = unitprice(diameter_small, price_small);
unitprice_large = unitprice(diameter_large, price_large);
```

where `unitprice_small` and `unitprice_large` are two variables of type *double*. One of the benefits of a function definition is that you can have multiple calls to the function in your program. This saves you the trouble of repeating the same (or almost the same) code. But we still must write the code for the function `unitprice`.

When we translate our pseudocode into C++ code, we obtain the following for the body of the function `unitprice`:

```
{//First draft of the function body for unitprice
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/2;
    area = PI * radius * radius;
    return (price/area);
}
```

Notice that we made PI a named constant using the modifier *const*. Also, notice the following line from the above code:

```
radius = diameter/2;
```

This is just a simple division by two, and you might think that nothing could be more routine. Yet, as written, this line contains a serious mistake. We want the division to produce the radius of the pizza including any fraction. For example, if we are considering buying the "bad luck special," which is a 13-inch pizza, then the radius is 6.5 inches. But the variable diameter is of type *int*. The constant 2 is also of type *int*. Thus, as we saw in Chapter 2, this line would perform integer division and would compute the radius 13/2 to be 6 instead of the correct value of 6.5, and we would have disregarded a half inch of pizza radius. In all likelihood this would go unnoticed, but the result could be that millions of subscribers to the Pizza Consumers Union could be wasting their money by buying the wrong size pizza. This is not likely to produce a major worldwide recession, but the program would be failing to accomplish its goal of helping consumers find the best buy. In a more important program, the result of such a simple mistake could be disastrous.

How do we fix this mistake? We want the division by two to be regular division that includes any fractional part in the answer. That form of division requires that at least one of the arguments to the division operator / must be of type *double*. We can use type casting to convert the constant 2 to a value of type *double*. Recall that *static_cast<double>(2)*, which is called a *type casting*, converts the *int* value 2 to a value of type *double*. Thus, if we replace 2 by *static_cast<double>(2)*, that will change the second argument in the division from type *int* to type *double* and the division will then produce the result we want. The rewritten assignment statement is:

static_cast <double>

```
radius = diameter/static_cast<double>(2);
```

The complete corrected code for the function definition of unitprice, along with the rest of the program, is shown in Display 3.9.

The type cast *static_cast<double>(2)* returns the value 2.0 so we could have used the constant 2.0 in place of *static_cast<double>(2)*. Either way, the function unitprice will return the same value. However, by using *static_cast<double>(2)* we make it conspicuously obvious that we want to do the version of division that includes the fractional part in its answer. If we instead used 2.0, then when revising or copying the code, we can easily make the mistake of changing 2.0 to 2, and that would produce a subtle problem.

We need to make one more remark about the coding of our program. As you can see in Display 3.9, when we coded tasks 4 and 5, we combined these two tasks into a single section of code consisting of a sequence of cout statements followed by an *if-else* statement. When two tasks are very simple and are closely related, it sometimes makes sense to combine them into a single task.

Display 3.9 Buying Pizza (*part 1 of 2*)


```
//Determines which of two pizza sizes is the best buy.
#include <iostream>
using namespace std;

double unitprice(int diameter, double price);
//Returns the price per square inch of a pizza. The formal
//parameter named diameter is the diameter of the pizza in inches.
//The formal parameter named price is the price of the pizza.

int main( )
{
    int diameter_small, diameter_large;
    double price_small, unitprice_small,
           price_large, unitprice_large;

    cout << "Welcome to the Pizza Consumers Union.\n";
    cout << "Enter diameter of a small pizza (in inches): ";
    cin >> diameter_small;
    cout << "Enter the price of a small pizza: $";
    cin >> price_small;
    cout << "Enter diameter of a large pizza (in inches): ";
    cin >> diameter_large;
    cout << "Enter the price of a large pizza: $";
    cin >> price_large;

    unitprice_small = unitprice(diameter_small, price_small);
    unitprice_large = unitprice(diameter_large, price_large);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Small pizza:\n"
         << "Diameter = " << diameter_small << " inches\n"
         << "Price = $" << price_small
         << " Per square inch = $" << unitprice_small << endl
         << "Large pizza:\n"
         << "Diameter = " << diameter_large << " inches\n"
         << "Price = $" << price_large
         << " Per square inch = $" << unitprice_large << endl;
```

Display 3.9 Buying Pizza (*part 2 of 2*)

```
    if (unitprice_large < unitprice_small)
        cout << "The large one is the better buy.\n";
    else
        cout << "The small one is the better buy.\n";
    cout << "Buon Appetito!\n";

    return 0;
}

double unitprice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/static_cast<double>(2);
    area = PI * radius * radius;
    return (price/area);
}
```

Sample Dialogue

```
Welcome to the Pizza Consumers Union.
Enter diameter of a small pizza (in inches): 10
Enter the price of a small pizza: $7.50
Enter diameter of a large pizza (in inches): 13
Enter the price of a large pizza: $14.75
Small pizza:
Diameter = 10 inches
Price = $7.50 Per square inch = $0.10
Large pizza:
Diameter = 13 inches
Price = $14.75 Per square inch = $0.11
The small one is the better buy.
Buon Appetito!
```

PROGRAM TESTING

Just because a program compiles and produces answers that look right does not mean the program is correct. In order to increase your confidence in your program you should test it on some input values for which you know the correct answer by some other means, such as working out the answer with paper and pencil or by using a handheld calculator. For example, it does not make sense to buy a two-inch pizza, but it can still be used as an easy test case for this program. It is an easy test case because it is easy to compute the answer by hand. Let's calculate the cost per square inch of a two-inch pizza that sells for $3.14. Since the diameter is two inches, the radius is one inch. The area of a pizza with radius one is `3.14159*1`^{2} which is `3.14159`. If we divide this into the price of $3.14, we find that the price per square inch is `3.14/3.14159`, which is approximately $1.00. Of course, this is an absurd size for a pizza and an absurd price for such a small pizza, but it is easy to determine the value that the function `unitprice` should return for these arguments.

Having checked your program on this one case, you can have more confidence in your program, but you still cannot be certain your program is correct. An incorrect program can sometimes give the correct answer, even though it will give incorrect answers on some other inputs. You may have tested an incorrect program on one of the cases for which the program happens to give the correct output. For example, suppose we had not caught the mistake we discovered when coding the function `unitprice`. Suppose we mistakenly used `2` instead of `static_cast<double>(2)` in the following line:

```
radius = diameter/static_cast<double>(2);
```

So that line reads as follows:

```
radius = diameter/2;
```

As long as the pizza diameter is an even number, like `2`, `8`, `10`, or `12`, the program gives the same answer whether we divide by `2` or by `static_cast<double>(2)`. It is unlikely that it would occur to you to be sure to check both even and odd size pizzas. However, if you test your program on several different pizza sizes, then there is a better chance that your test cases will contain samples of the relevant kinds of data.

Programming TIP

Use Pseudocode

pseudocode

Algorithms are typically expressed in *pseudocode*. **Pseudocode** is a mixture of C++ (or whatever programming language you are using) and ordinary English (or

whatever human language you are using). Pseudocode allows you to state your algorithm precisely without having to worrying about all the details of C++ syntax. When the C++ code for a step in your algorithm is obvious, there is little point in stating it in English. When a step is difficult to express in C++, the algorithm will be clearer if the step is expressed in English. You can see an example of pseudocode in the previous case study, where we expressed our algorithm for the function `unitprice` in pseudocode.

SELF-TEST EXERCISES

11 What is the purpose of the comment that accompanies a function declaration?

12 What is the principle of procedural abstraction as applied to function definitions?

13 What does it mean when we say the programmer who uses a function should be able to treat the function like a black box? (*Hint:* This question is very closely related to the previous question.)

14 Carefully describe the process of program testing.

15 Consider two possible definitions for the function `unitprice`. One is the definition given in Display 3.9. The other definition is the same except that the type cast `static_cast<double>(2)` is replaced with the constant `2.0`, in other words, the line

```
radius = diameter/static_cast<double>(2);
```

is replaced with the line

```
radius = diameter/2.0;
```

Are these two possible function definitions black-box equivalent?

3.5 Local Variables

He was a local boy,
not known outside his home town.

COMMON SAYING

In the last section we advocated using functions as if they were black boxes. In order to define a function so that it can be used as a black box, you often need to give the function variables of its own that do not interfere with the rest of your program.

These variables that "belong to" a function are called *local variables*. In this section we describe local variables and tell you how to use them.

The Small Program Analogy

Look back at the program in Display 3.1. It includes a call to the predefined function `sqrt`. We did not need to know anything about the details of the function definition for `sqrt` in order to use this function. In particular, we did not need to know what variables were declared in the definition of `sqrt`. A function that you define is no different. Variable declarations in function definitions that you write are as separate as those in the function definitions for the predefined functions. Variable declarations within a function definition are the same as if they were variable declarations in another program. If you declare a variable in a function definition and then declare another variable of the same name in the `main` part of your program (or in the body of some other function definition), then these two variables are two different variables, even though they have the same name. Let's look at a program that does have a variable in a function definition with the same name as another variable in the program.

The program in Display 3.10 has two variables named `average_pea`; one is declared and used in the function definition for the function `est_total`, and the other is declared and used in the `main` part of the program. The variable `average_pea` in the function definition for `est_total` and the variable `average_pea` in the `main` part of the program are two different variables. It is the same as if the function `est_total` were a predefined function. The two variables named `average_pea` will not interfere with each other any more than two variables in two completely different programs would. When the variable `average_pea` is given a value in the function call to `est_total`, this does not change the value of the variable in the `main` part of the program that is also named `average_pea`. (The details of the program in Display 3.10, other than this coincidence of names, are explained in the Programming Example section that follows this section.)

local to a function

scope

local variable

Variables that are declared within the body of a function definition are said to be **local to that function** or to have that function as their **scope.** Variables that are defined within the `main` body of the program are said to be **local to the `main` part of the program** or to have the `main` part of the program as their **scope.** There are other kinds of variables that are not local to any function or to the `main` part of the program, but we will have no use for such variable. Every variable we will use is either local to a function definition or local to the `main` part of the program. When we say that a variable is a **local variable** without any mention of a function and without any mention of the `main` part of the program, we mean that the variable is local to some function definition.

Display 3.10 Local Variables (*part 1 of 2*)


```
//Computes the average yield on an experimental pea growing patch.
#include <iostream>
using namespace std;

double est_total(int min_peas, int max_peas, int pod_count);
//Returns an estimate of the total number of peas harvested.
//The formal parameter pod_count is the number of pods.
//The formal parameters min_peas and max_peas are the minimum
//and maximum number of peas in a pod.

int main( )
{
    int max_count, min_count, pod_count;
    double average_pea, yield;

    cout << "Enter minimum and maximum number of peas in a pod: ";
    cin >> min_count >> max_count;
    cout << "Enter the number of pods: ";
    cin >> pod_count;
    cout << "Enter the weight of an average pea (in ounces): ";
    cin >> average_pea;

    yield =
          est_total(min_count, max_count, pod_count) * average_pea;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(3);
    cout << "Min number of peas per pod = " << min_count << endl
         << "Max number of peas per pod = " << max_count << endl
         << "Pod count = " << pod_count << endl
         << "Average pea weight = "
         << average_pea << " ounces" << endl
         << "Estimated average yield = " << yield << " ounces"
         << endl;

    return 0;
}
```

This variable named average_pea *is local to the* main *part of the program.*

Display 3.10 Local Variables (*part 2 of 2*)

```
double est_total(int min_peas, int max_peas, int pod_count)
{
    double average_pea;

    average_pea = (max_peas + min_peas)/2.0;
    return (pod_count * average_pea);
}
```

This variable named average_pea *is local to the function est_total.*

Sample Dialogue

```
Enter minimum and maximum number of peas in a pod: 4 6
Enter the number of pods: 10
Enter the weight of an average pea (in ounces): 0.5
Min number of peas per pod = 4
Max number of peas per pod = 6
Pod count = 10
Average pea weight = 0.500 ounces
Estimated average yield = 25.000 ounces
```

Local Variables

Variables that are declared within the body of a function definition are said to be **local to that function** or to have that function as their **scope.** Variables that are declared within the `main` part of the program are said to be **local to the `main` part of the program** or to have the main part of the program as their **scope.** When we say that a variable is a **local variable** without any mention of a function and without any mention of the `main` part of the program, we mean that the variable is local to some function definition. If a variable is local to a function, then you can have another variable with the same name that is declared in the `main` part of the program or in another function definition, and these will be two different variables, even though they have the same name.

Programming EXAMPLE
Experimental Pea Patch

The program in Display 3.10 gives an estimate for the total yield on a small garden plot used to raise an experimental variety of peas. The function `est_total` returns an estimate of the total number of peas harvested. The function `est_total` takes three arguments. One argument is the number of pea pods that were harvested. The other two arguments are used to estimate the average number of peas in a pod. Different pea pods contain differing numbers of peas so the other two arguments to the function are the smallest and the largest number of peas that were found in any one pod. The function `est_total` averages these two numbers and uses this average as an estimate for the average number of peas in a pod.

Global Constants and Global Variables

As we noted in Chapter 2, you can and should name constant values using the *const* modifier. For example, in Display 3.9 we used the following declaration to give the name `PI` to the constant `3.14159`:

```
const double PI = 3.14159;
```

In Display 3.3, we used the *const* modifier to give a name to the rate of sales tax with the following declaration:

```
const double TAX_RATE = 0.05; //5% sales tax
```

As with our variable declarations, we placed these declarations for naming constants inside the body of the functions that used them. This worked out fine because each named constant was used by only one function. However, it can easily happen that more than one function uses a named constant. In that case you can place the declaration for naming a constant at the beginning of your program, outside of the body of all the functions and outside the body of the `main` part of your program. The named constant is then said to be a **global named constant** and the named constant can be used in any function definition that follows the constant declaration.

Display 3.11 shows a program with an example of a global named constant. The program asks for a radius and then computes both the area of a circle and the volume of a sphere with that radius. The programmer who wrote that program looked up the formulas for computing those quantities and found the following:

$$area = \pi \times (radius)^2$$
$$volume = (4/3) \times \pi \times (radius)^3$$

Both formulas include the constant π, which is approximately equal to 3.14159. The symbol π is the Greek letter called "pi." In previous programs we have used the following declaration to produce a named constant called `PI` to use when we convert such formulas to C++ code:

```
const double PI = 3.14159;
```

In the program in Display 3.11 we use the same declaration but place it near the beginning of the file, so that it defines a global named constant that can be used in all the function bodies.

The compiler allows you wide latitude in where you place the declarations for your global named constants, but to aid readability you should place all your `include` directives together, all your global named constant declarations together in another group, and all your function declarations together. We will follow standard practice and place all our global named constant declarations after our `include` directives and before our function declarations.

Placing all named constant declarations at the start of your program can aid readability even if the named constant is used by only one function. If the named constant might need to be changed in a future version of your program, it will be easier to find if it is at the beginning of your program. For example, placing the constant declaration for the sales tax rate at the beginning of an accounting program will make it easy to revise the program should the tax rate increase.

global variables

It is possible to declare ordinary variables, without the *`const`* modifier, as **global variables,** which are accessible to all function definitions in the file. This is done the same way that it is done for global named constants, except that the modifier *`const`* is not used in the variable declaration. However, there is seldom any need to use such global variables. Moreover, global variables can make a program harder to understand and maintain, so we will not use any global variables. Once you have had more experience designing programs, you may choose to occasionally use global variables.

Call-by-Value Formal Parameters Are Local Variables

Formal parameters are more than just blanks that are filled in with the argument values for the function. Formal parameters are actually variables that are local to the function definition, so they can be used just like a local variable that is declared in the function definition. Earlier in this chapter we described the call-by-value mechanism which handles the arguments in a function call. We can now define this mechanism for "plugging in arguments" in more detail. When a function is called, the formal parameters for the function (which are local variables) are initialized to the values of the arguments. This is the precise meaning of the phrase "plugged in for the formal parameters" which we have been using. Typically, a formal parameter is used only as a kind of blank, or placeholder, that is filled in by the value of its corresponding argument; occasionally, however, a formal parameter is used as a variable whose value is changed. In this section we will give one example of a formal parameter used as a local variable.

Display 3.11 A Global Named Constant (*part 1 of 2*)


```
//Computes the area of a circle and the volume of a sphere.
//Uses the same radius for both calculations.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;

double area(double radius);
//Returns the area of a circle with the specified radius.

double volume(double radius);
//Returns the volume of a sphere with the specified radius.

int main( )
{
    double radius_of_both, area_of_circle, volume_of_sphere;

    cout << "Enter a radius to use for both a circle\n"
         << "and a sphere (in inches): ";
    cin >> radius_of_both;

    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);

    cout << "Radius = " << radius_of_both << " inches\n"
         << "Area of circle = " << area_of_circle
         << " square inches\n"
         << "Volume of sphere = " << volume_of_sphere
         << " cubic inches\n";

    return 0;
}
```

Display 3.11 A Global Named Constant (*part 2 of 2*)

```
double area(double radius)
{
    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

Sample Dialogue

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```

The program in Display 3.12 is the billing program for the law offices of Dewey, Cheatham, and Howe. Notice that, unlike other law firms, the firm of Dewey, Cheatham, and Howe does not charge for any time less than a quarter of an hour. That is why it's called "the law office with a heart." If they work for one hour and fourteen minutes, they only charge for four quarter hours, not five quarter hours as other firms do; so you would pay only $600 for the consultation.

Notice the formal parameter `minutes_worked` in the definition of the function `fee`. It is used as a variable and has its value changed by the following line, which occurs within the function definition:

```
minutes_worked = hours_worked*60 + minutes_worked;
```

Do not add a declaration for a formal parameter.

Formal parameters are local variables just like the variables you declare within the body of a function. However, you should not add a variable declaration for the formal parameters. Listing the formal parameter `minutes_worked` in the function declaration also serves as the variable declaration. The following is the *wrong way* to start the function definition for `fee` as it declares `minutes_worked` twice:

```
double fee(int hours_worked, int minutes_worked)
{
    int quarter_hours;
    int minutes_worked;
          . . .
```

Do NOT do this!

Display 3.12 Formal Parameter Used as a Local Variable (*part 1 of 2*)


```
//Law office billing program.
#include <iostream>
using namespace std;

const double RATE = 150.00; //Dollars per quarter hour.

double fee(int hours_worked, int minutes_worked);
//Returns the charges for hours_worked hours and
//minutes_worked minutes of legal services.

int main()
{
    int hours, minutes;
    double bill;

    cout << "Welcome to the offices of\n"
         << "Dewey, Cheatham, and Howe.\n"
         << "The law office with a heart.\n"
         << "Enter the hours and minutes"
         << " of your consultation:\n";
    cin >> hours >> minutes;

    bill = fee(hours, minutes);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For " << hours << " hours and " << minutes
         << " minutes, your bill is $" << bill << endl;

    return 0;
}

double fee(int hours_worked, int minutes_worked)
{
    int quarter_hours;

    minutes_worked = hours_worked*60 + minutes_worked;
    quarter_hours = minutes_worked/15;
    return (quarter_hours*RATE);
}
```

The value of `minutes` *is not changed by the call to* `fee`.

`minutes_worked` *is a local variable initialized to the value of* `minutes`.

Display 3.12 Formal Parameter Used as a Local Variable (*part 2 of 2*)

Sample Dialogue

```
Welcome to the offices of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
2 45
For 2 hours and 45 minutes, your bill is $1650.00
```

Namespaces Revisited

Thus far, we have started all of our programs with the following two lines:

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

However, the start of the file is not always the best location for the line

```
using namespace std;
```

We will eventually be using more namespaces than just `std`. In fact, we may be using different namespaces in different function definitions. If you place the directive

```
using namespace std;
```

inside the brace { that starts the body of a function definition, then the *`using`* directive applies to only that function definition. This will allow you to use two different namespaces in two different function definitions, even if the two function definitions are in the same file and even if the two namespaces have some name(s) with different meanings in the two different namespaces.

Placing a *`using`* directive inside a function definition is analogous to placing a variable declaration inside a function definition. If you place a variable definition inside a function definition, the variable is local to the function; that is, the meaning of the variable declaration is confined to the function definition. If you place a *`using`* directive inside a function definition, the *`using`* directive is local to the function definition; in other words, the meaning of the *`using`* directive is confined to the function definition.

It will be some time before we use any namespace other than `std` in a *`using`* directive, but it will be good practice to start placing these *`using`* directives where they should go.

In Display 3.13 we have rewritten the program in Display 3.11 with the *`using`* directives where they should be placed. The program in Display 3.13 will behave exactly the same as the one in Display 3.11. In this particular case, the difference is only one of style, but when you start to use more namespaces, the difference will affect how your programs perform.

SELF-TEST EXERCISES

16 If you use a variable in a function definition, where should you declare the variable? In the function definition? In the `main` part of the program? Anyplace that is convenient?

17 Suppose a function named `Function1` has a variable named `Sam` declared within the definition of `Function1`, and a function named `Function2` also has a variable named `Sam` declared within the definition of `Function2`. Will the program compile (assuming everything else is correct)? If the program will compile, will it run (assuming that everything else is correct)? If it runs, will it generate an error message when run (assuming everything else is correct)? If it runs and does not produce an error message when run, will it give the correct output (assuming everything else is correct)?

18 The following function is supposed to take as arguments a length expressed in feet and inches and return the total number of inches in that many feet and inches. For example, `total_inches(1, 2)` is supposed to return 14, because 1 foot and 2 inches is the same as 14 inches. Will the following function perform correctly? If not, why not?

```
double total_inches(int feet, int inches)
{
    inches = 12*feet + inches;
    return inches;
}
```

19 Write a function declaration and function definition for a function called `read_filter` that has no parameters and that returns a value of type *`double`*. The function `read_filter` prompts the user for a value of type *`double`* and reads the value into a local variable. The function returns the value read in provided this value is greater than or equal to zero and returns zero if the value read in is negative.

Display 3.13 Using Namespaces (*part 1 of 2*)


```
//Computes the area of a circle and the volume of a sphere.
//Uses the same radius for both calculations.
#include <iostream>
#include <cmath>

const double PI = 3.14159;

double area(double radius);
//Returns the area of a circle with the specified radius.

double volume(double radius);
//Returns the volume of a sphere with the specified radius.

int main( )
{
    using namespace std;

    double radius_of_both, area_of_circle, volume_of_sphere;

    cout << "Enter a radius to use for both a circle\n"
         << "and a sphere (in inches): ";
    cin >> radius_of_both;

    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);

    cout << "Radius = " << radius_of_both << " inches\n"
         << "Area of circle = " << area_of_circle
         << " square inches\n"
         << "Volume of sphere = " << volume_of_sphere
         << " cubic inches\n";

    return 0;
}
```

Display 3.13 Using Namespaces (*part 2 of 2*)

```
double area(double radius)
{
    using namespace std;

    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    using namespace std;

    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

The sample dialogue for this program would be the same as the one for the program in Display 3.11.

Programming EXAMPLE
The Factorial Function

Display 3.14 contains the function declaration and definition for a commonly used mathematical function known as the *factorial* function. In mathematics texts, the factorial function is usually written *n*! and is defined to be the product of all the integers from 1 to *n*. In traditional mathematical notation, you can define *n*! as follows:

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

In the function definition we perform the multiplication with a `while` loop. Note that the multiplication is performed in the reverse order to what you might expect. The program multiplies by `n`, then `n`–1, then `n`–2, and so forth.

formal parameter used as a local variable

The function definition for `factorial` uses two local variables: `product`, which is declared at the start of the function body, and the formal parameter `n`. Since a formal parameter is a local variable, we can change its value. In this case we change the value of the formal parameter `n` with the decrement operator `n--`. (The decrement operator was discussed in Chapter 2.)

Display 3.14 Factorial Function

Function Declaration

```
int factorial(int n);
//Returns factorial of n.
//The argument n should be nonnegative.
```

Function Definition

```
int factorial(int n)
{
    int product = 1;
    while (n > 0)
    {
        product = n * product;
        n--;
    }

    return product;
}
```

formal parameter n *used as a local variable*

Each time the body of the loop is executed, the value of the variable `product` is multiplied by the value of `n`, and then the value of `n` is decreased by one using `n--`. If the function `factorial` is called with `3` as its argument, then the first time the loop body is executed the value of `product` is `3`, the next time the loop body is executed the value of `product` is `3*2`, the next time the value of `product` is `3*2*1`, and then the *`while`* loop ends. Thus, the following will set the variable `x` equal to `6` which is `3*2*1`:

```
x = factorial(3);
```

Notice that the local variable `product` is initialized to the value `1` when the variable is declared. (This way of initializing a variable when it is declared was introduced in Chapter 2.) It is easy to see that `1` is the correct initial value for the variable `product`. To see that this is the correct initial value for `product` note that, after executing the body of the *`while`* loop the first time, we want the value of `product` to be equal to the (original) value of the formal parameter `n`; if `product` is initialized to `1`, then this will be what happens.

3.6 Overloading Function Names

"...—and that shows that there are three hundred and sixty-four days when you might get un-birthday presents—"
"Certainly," said Alice.
"And only one *for birthday presents, you know. There's glory for you!"*
"I don't know what you mean by 'glory,' " Alice said.
Humpty Dumpty smiled contemptuously, "Of course you don't—till I tell you. I mean 'there's a nice knock-down argument for you!' "
"But 'glory' doesn't mean 'a nice knock-down argument,' " Alice objected.
"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."
"The question is," said Alice, "whether you can *make words mean so many different things."*
"The question is," said Humpty Dumpty, "which is to be master—that's all."

LEWIS CARROLL, *THROUGH THE LOOKING-GLASS*

C++ allows you to give two or more different definitions to the same function name, which means you can reuse names that have strong intuitive appeal across a variety of situations. For example, you could have three functions called `max`: one that computes the largest of two numbers, another that computes the largest of three numbers, and yet another that computes the largest of four numbers. When you give two (or more) function definitions for the same function name, that is called **overloading** the function name. Overloading does require some extra care in defining your functions, and should not be used unless it will add greatly to your program's readability. But when it is appropriate, overloading can be very effective.

Introduction to Overloading

Suppose you are writing a program that requires you to compute the average of two numbers. You might use the following function definition:

```
double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}
```

Now suppose your program also requires a function to compute the average of three numbers. You might define a new function called `ave3` as follows:

```
double ave3(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

This will work, and in many programming languages you have no choice but to do something like this. Fortunately, C++ allows for a more elegant solution. In C++ you can simply use the same function name ave for both functions. In C++ you can use the following function definition in place of the function definition ave3:

```
double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

so that the function name ave then has two definitions. This is an example of overloading. In this case we have overloaded the function name ave. In Display 3.15 we have embedded these two function definitions for ave into a complete sample program. Be sure to notice that each function definition has its own function declaration.

Overloading is a great idea. It makes a program easier to read. It saves you from going crazy trying to think up a new name for a function just because you already used the most natural name in some other function definition. But how does the compiler know which function definition to use when it encounters a call to a function name that has two or more definitions? The compiler cannot read a programmer's mind. In order to tell which function definition to use, the compiler checks the number of arguments and the types of the arguments in the function call. In the program in Display 3.15, one of the functions called ave has two arguments and the other has three arguments. To tell which definition to use, the compiler simply counts the number of arguments in the function call. If there are two arguments, it uses the first definition. If there are three arguments, it uses the second definition.

determining which definition applies

Whenever you give two or more definitions to the same function name, the various function definitions must have different specifications for their arguments; that is, any two function definitions that have the same function name must use different numbers of formal parameters or use formal parameters of different types (or both). Notice that when you overload a function name, the function declarations for the two different definitions must differ in their formal parameters. *You cannot overload a function name by giving two definitions that differ only in the type of the value returned.*

Overloading a Function Name

If you have two or more function definitions for the same function name, that is called **overloading.** When you overload a function name, the function definitions must have different numbers of formal parameters or some formal parameters of different types. When there is a function call, the compiler uses the function definition whose number of formal parameters and types of formal parameters match the arguments in the function call.

Display 3.15 Overloading a Function Name


```
//Illustrates overloading the function name ave.
#include <iostream>

double ave(double n1, double n2);
//Returns the average of the two numbers n1 and n2.

double ave(double n1, double n2, double n3);
//Returns the average of the three numbers n1, n2, and n3.

int main( )
{
    using namespace std;
    cout << "The average of 2.0, 2.5, and 3.0 is "
         << ave(2.0, 2.5, 3.0) << endl;

    cout << "The average of 4.5 and 5.5 is "
         << ave(4.5, 5.5) << endl;

    return 0;
}

double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}

double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

two arguments

three arguments

Output

```
The average of 2.0, 2.5, and 3.0 is 2.50000
The average of 4.5 and 5.5 is 5.00000
```

Overloading is not really new to you. You saw a kind of overloading in Chapter 2 with the division operator /. If both operands are of type *int*, as in 13/2, then the value returned is the result of integer division, in this case 6. On the other hand, if one or both operands are of type *double*, then the value returned is the result of regular division; for example, 13/2.0 returned the value 6.5. There are two definitions for the division operator /, and the two definitions are distinguished not by having different numbers of operands, but rather by requiring operands of different types. The difference between overloading of / and overloading function names is that the compiler has already done the overloading of / and we program the overloading of the function name. We will see in a later chapter how to overload operators such as +, −, and so on.

Programming EXAMPLE
Revised Pizza-Buying Program

The Pizza Consumers Union has been very successful with the program that we wrote for it in Display 3.9. In fact, now everybody always buys the pizza that is the best buy. One disreputable pizza parlor used to make money by fooling consumers into buying the more expensive pizza, but our program has put an end to their evil practices. However, the owners wish to continue their despicable behavior and have come up with a new way to fool consumers. They now offer both round pizzas and rectangular pizzas. They know that the program we wrote cannot deal with rectangularly shaped pizzas, so they hope they can again confuse consumers. We need to update our program so that we can foil their nefarious scheme. We want to change the program so that it can compare a round pizza and a rectangular pizza.

The changes we need to make to our pizza evaluation program are clear: We need to change the input and output a bit so that it deals with two different shapes of pizzas. We also need to add a new function that can compute the cost per square inch of a rectangular pizza. We could use the following function definition in our program so that we can compute the unit price for a rectangular pizza:

```
double unitprice_rectangular
        (int length, int width, double price)
{
    double area = length * width;
    return (price/area);
}
```

However, this is a rather long name for a function; in fact, it's so long that we needed to put the function heading on two lines. That is legal, but it would be nicer to use the

same name, unitprice, for both the function that computes the unit price for a round pizza and for the function that computes the unit price for a rectangular pizza. Since C++ allows overloading of function names, we can do this. Having two definitions for the function unitprice will pose no problems to the compiler because the two functions will have different numbers of arguments. Display 3.16 shows the program we obtained when we modified our pizza evaluation program to allow us to compare round pizzas with rectangular pizzas.

Automatic Type Conversion

Suppose that the following function definition occurs in your program, and that you have *not* overloaded the function name mpg (so this is the only definition of a function called mpg).

```
double mpg(double miles, double gallons)
//Returns miles per gallon.
{
    return (miles/gallons);
}
```

If you call the function mpg with arguments of type *int*, then C++ will automatically convert any argument of type *int* to a value of type *double*. Hence, the following will output 22.5 miles per gallon to the screen:

```
cout << mpg(45, 2) << " miles per gallon";
```

C++ converts the 45 to 45.0 and the 2 to 2.0, then performs the division 45.0/2.0 to obtain the value returned, which is 22.5.

interaction of overloading and type conversion

If a function requires an argument of type *double* and you give it an argument of type *int*, C++ will automatically convert the *int* argument to a value of type *double*. This is so useful and natural that we hardly give it a thought. However, overloading can interfere with this automatic type conversion. Let's look at an example.

Suppose you had (foolishly) overloaded the function name mpg so that your program also contained the following definition of mpg (as well as the one above):

```
int mpg(int goals, int misses)
//Returns the Measure of Perfect Goals
//which is computed as (goals - misses).
{
    return (goals – misses);
}
```

Display 3.16 Overloading a Function Name (*part 1 of 3*)


```
//Determines whether a round pizza or a rectangular pizza is the best buy.
#include <iostream>

double unitprice(int diameter, double price);
//Returns the price per square inch of a round pizza.
//The formal parameter named diameter is the diameter of the pizza
//in inches. The formal parameter named price is the price of the pizza.

double unitprice(int length, int width, double price);
//Returns the price per square inch of a rectangular pizza
//with dimensions length by width inches.
//The formal parameter price is the price of the pizza.

int main( )
{
    using namespace std;
    int diameter, length, width;
    double price_round, unit_price_round,
           price_rectangular, unitprice_rectangular;

    cout << "Welcome to the Pizza Consumers Union.\n";
    cout << "Enter the diameter in inches"
         << " of a round pizza: ";
    cin >> diameter;
    cout << "Enter the price of a round pizza: $";
    cin >> price_round;
    cout << "Enter length and width in inches\n"
         << "of a rectangular pizza: ";
    cin >> length >> width;
    cout << "Enter the price of a rectangular pizza: $";
    cin >> price_rectangular;

    unitprice_rectangular =
              unitprice(length, width, price_rectangular);
    unit_price_round = unitprice(diameter, price_round);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
```

Display 3.16 Overloading a Function Name (*part 2 of 3*)

```
    cout << endl
         << "Round pizza: Diameter = "
         << diameter << " inches\n"
         << "Price = $" << price_round
         << " Per square inch = $" << unit_price_round
         << endl
         << "Rectangular pizza: Length = "
         << length << " inches\n"
         << "Rectangular pizza: Width = "
         << width << " inches\n"
         << "Price = $" << price_rectangular
         << " Per square inch = $" << unitprice_rectangular
         << endl;

    if (unit_price_round < unitprice_rectangular)
        cout << "The round one is the better buy.\n";
    else
        cout << "The rectangular one is the better buy.\n";
    cout << "Buon Appetito!\n";

    return 0;
}

double unitprice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/static_cast<double>(2);
    area = PI * radius * radius;
    return (price/area);
}

double unitprice(int length, int width, double price)
{
    double area = length * width;
    return (price/area);
}
```

Display 3.16 Overloading a Function Name (*part 3 of 3*)

Sample Dialogue

```
Welcome to the Pizza Consumers Union.
Enter the diameter in inches of a round pizza: 10
Enter the price of a round pizza: $8.50
Enter length and width in inches
of a rectangular pizza: 6 4
Enter the price of a rectangular pizza: $7.55

Round pizza: Diameter = 10 inches
Price = $8.50 Per square inch = $0.11
Rectangular pizza: Length = 6 inches
Rectangular pizza: Width = 4 inches
Price = $7.55 Per square inch = $0.31
The round one is the better buy.
Buon Appetito!
```

In a program that contains both of these definitions for the function name `mpg`, the following will (unfortunately) output `43 miles per gallon` (since 43 is 45 – 2):

```
cout << mpg(45, 2) << " miles per gallon";
```

When C++ sees the function call `mpg(45, 2)`, which has two arguments of type `int`, C++ *first* looks for a function definition of `mpg` that has two formal parameters of type `int`. If it finds such a function definition, C++ uses that function definition. C++ does not convert an `int` argument to a value of type `double` unless that is the only way it can find a matching function definition.

The `mpg` example illustrates one more point about overloading. You should not use the same function name for two unrelated functions. Such careless use of function names is certain to eventually produce confusion.

SELF-TEST EXERCISES

20 Suppose you have two function definitions with the following function declarations:

```
double score(double time, double distance);
int score(double points);
```

Which function definition would be used in the following function call and why would it be the one used? (x is of type *double*.)

```
final_score = score(x);
```

21 Suppose you have two function definitions with the following function declarations:

```
double the_answer(double data1, double data2);
double the_answer(double time, int count);
```

Which function definition would be used in the following function call and why would it be the one used? (x and y are of type *double*.)

```
x = the_answer(y, 6.0);
```

22 Suppose you have two function definitions with the function declarations given in Self-Test Exercise 21.Which function definition would be used in the following function call and why would it be the one used?

```
x = the_answer(5, 6);
```

23 Suppose you have two function definitions with the function declarations given in Self-Test Exercise 21.Which function definition would be used in the following function call and why would it be the one used?

```
x = the_answer(5, 6.0);
```

24 This question has to do with the Programming Example "Revised Pizza-Buying Program." Suppose the evil pizza parlor that is always trying to fool customers introduces a square pizza. Can you overload the function `unitprice` so that it can compute the price per square inch of a square pizza as well as the price per square inch of a round pizza? Why or why not?

25 Look at the program in Display 3.16. The `main` function contains the *using* directive:

```
using namespace std;
```

Why doesn't the method `unitprice` contain this *using* directive?

CHAPTER SUMMARY

- A good plan of attack for designing the algorithm for a program is to break down the task to be accomplished into a few subtasks, then decompose each subtask into smaller subtasks, and so forth until the subtasks are simple enough that they can easily be implemented as C++ code. This approach is called **top-down design.**
- A function that returns a value is like a small program. The arguments to the function serve as the input to this "small program" and the value returned is like the output of the "small program."
- When a subtask for a program takes some values as input and produces a single value as its only result, then that subtask can be implemented as a function.
- A function should be defined so that it can be used as a black box. The programmer who uses the function should not need to know any details about how the function is coded. All the programmer should need to know is the function declaration and the accompanying comment that describes the value returned. This rule is sometimes called the **principle of procedural abstraction.**
- A variable that is declared in a function definition is said to be **local to the function.**
- Global named constants are declared using the `const` modifier. Declarations for global named constants are normally placed at the start of a program after the `include` directives and before the function declarations.
- Call-by-value formal parameters (which are the only kind of formal parameter discussed in this chapter) are variables that are local to the function. Occasionally, it is useful to use a formal parameter as a local variable.
- When you have two or more function definitions for the same function name, that is called **overloading** the function name. When you overload a function name, the function definitions must have different numbers of formal parameters or some formal parameters of different types.

Answers to Self-Test Exercises

1
```
4.0     4.0     8.0
8.0     8.0     1.21
3       3       0
3.0     3.5     3.5
6.0     6.0     5.0
5.0     4.5     4.5
3       3.0     3.0
```

2 `sqrt(x + y)`, `pow(x, y + 7)`, `sqrt(area + fudge)`, `sqrt(time+tide)/nobody`, `(–b + sqrt(b*b – 4*a*c))/(2*a)`, `abs(x – y)` or `labs(x – y)` or `fabs(x – y)`

3
```
//Computes the square root of 3.14159.
#include <iostream>
#include <cmath>//provides sqrt and PI.
using namespace std;
int main( )
{
    cout << "The square root of  " >> PI
         << sqrt(PI) << endl;
    return 0;
}
```

4 a.
```
//To determine whether the compiler will tolerate
//spaces before the # in the #include:
  #include <iostream>
int main( )
{
     cout << "hello world" << endl;
     return 0;
}
```

b.
```
//To determine if the compiler will allow spaces
//between the # and include in the #include:
# include<iostream>
using namespace std;
//The rest of the program can be identical to the above.
```

5 `Wow`

6 The function declaration is:

```
int sum(int n1, int n2, int n3);
//Returns the sum of n1, n2, and n3.
```

The function definition is:

```
int sum(int n1, int n2, int n3)
{
    return (n1 + n2 + n3);
}
```

7 The function declaration is:

```
double ave(int n1, double n2);
//Returns the average of n1 and n2.
```

The function definition is:

```
double ave(int n1, double n2)
{
    return ((n1 + n2)/2.0);
}
```

8 The function declaration is:

```
char positive_test(double number);
//Returns 'P' if number is positive.
//Returns 'N' if number is negative or zero.
```

The function definition is:

```
char positive_test(double number)
{
    if (number > 0)
        return 'P';
    else
        return 'N';
}
```

9 Suppose the function is defined with arguments, say `param1` and `param2`. The function is then called with corresponding arguments `arg1` and `arg2`. The values of the arguments are "plugged in" for the corresponding formal parameters, `arg1` into `param1`, `arg2` into `param2`. The formal parameters are then used in the function.

10 Predefined (library) functions usually require that you `#include` a header file. For a programmer-defined function, the programmer puts the code for the function either into the file with the main part of the program or in another file to be compiled and linked to the main program.

11 The comment explains what value the function returns and gives any other information that you need to know in order to use the function.

12 The principle of procedural abstraction says that a function should be written so that it can be used like a black box. This means that the programmer who uses the function need not look at the body of the function definition to see how the function works. The function declaration and accompanying comment should be all the programmer needs to know in order to use the function.

13 When we say that the programmer who uses a function should be able to treat the function like a black box, we mean the programmer should not need to look at the body of the function definition to see how the function works. The function declaration and accompanying comment should be all the programmer needs to know in order to use the function.

14 In order to increase your confidence in your program, you should test it on input values for which you know the correct answers. Perhaps you can calculate the answers by some other means, such as pencil and paper or hand calculator.

15 Yes, the function would return the same value in either case, so the two definitions are black-box equivalent.

16 If you use a variable in a function definition, you should declare the variable in the body of the function definition.

17 Everything will be fine. The program will compile (assuming everything else is correct). The program will run (assuming that everything else is correct). The program will not generate an error message when run (assuming everything else is correct). The program will give the correct output (assuming everything else is correct).

18 The function will work fine. That is the entire answer, but here is some additional information: The formal parameter `inches` is a call-by-value parameter and, as discussed in the text, it is therefore a local variable. Thus, the value of the argument will not be changed.

19 The function declaration is:

```
double read_filter( );
//Reads a number from the keyboard. Returns the number
//read provides it is >= 0; otherwise returns zero.
```

The function definition is:

```
//uses iostream
double read_filter( )
{
    using namespace std;
    double value_read;
    cout << "Enter a number:\n";
    cin >> value_read;

    if (value_read >= 0)
        return value_read;
    else
        return 0.0;
}
```

20 The function call has only one argument, so it would use the function definition that has only one formal parameter.

21 The function call has two arguments of type `double`, so it would use the function corresponding to the function declaration with two arguments of type `double` (that is, the first function declaration).

22 The second argument is of type `int` and the first argument would be automatically converted to type `double` by C++ if needed, so it would use the function corresponding to the function declaration with the first arguments of type `double` and the second argument of type `int` (that is, the second function declaration).

23 The second argument is of type `double` and the first argument would be automatically converted to type `double` by C++ if needed, so it would use the function corresponding to the function declaration with two arguments of type `double` (that is, the first function declaration).

24 This cannot be done (at least not in any nice way). The natural ways to represent a square and a round pizza are the same. Each is naturally represented as one number, which is the diameter for a round pizza and the length of a side for a square pizza. In either case the function `unitprice` would need to have one formal parameter of type `double` for the price and one formal parameter of type `int` for the size (either radius or side). Thus, the two function declarations would have the same number and types of formal parameters. (Specifically, they would both have one formal parameter of type `double` and one formal parameter of type `int`.) Thus, the compiler would not be able to decide which definition to use. You can still defeat this evil pizza parlor's strategy by defining two functions, but they will need to have different names.

25 The definition of `unitprice` does not do any input or output and so does not use the library `iostream`. In `main` we needed the `using` directive because `cin` and `cout` are defined in `iostream` and those definitions place `cin` and `cout` in the `std` namespace.

Programming Projects

1 A liter is 0.264179 gallons. Write a program that will read in the number of liters of gasoline consumed by the user's car and the number of miles traveled by the car, and will then output the number of miles per gallon the car delivered. Your program should allow the user to repeat this calculation as often as the user wishes. Define a function to compute the number of miles per gallon. Your program should use a globally defined constant for the number of liters per gallon.

2 The price of stocks is sometimes given to the nearest eighth of a dollar; for example, 297/8 or 891/2. Write a program that computes the value of the user's holding of one stock. The program asks for the number of shares of stock owned, the whole dollar portion of the price and the fraction portion. The fraction portion is to be input as two *int* values, one for the numerator and one for the denominator. The program then outputs the value of the user's holdings. Your program should allow the user to repeat this calculation as often as the user wishes. Your program will include a function definition that has three *int* arguments consisting of the whole dollar portion of the price and the two integers that make up the fraction part. The function returns the price of one share of stock as a single number of type *double*.

3 Write a program to gauge the rate of inflation for the past year. The program asks for the price of an item (such as a hot dog or a one carat diamond) both one year ago and today. It estimates the inflation rate as the difference in price divided by the year ago price. Your program should allow the user to repeat this calculation as often as the user wishes. Define a function to compute the rate of inflation. The inflation rate should be a value of type *double* giving the rate as a percent, for example 5.3 for 5.3%.

4 Enhance your program from the previous exercise by having it also print out the estimated price of the item in one and in two years from the time of the calculation. The increase in cost over one year is estimated as the inflation rate times the price at the start of the year. Define a second function to determine the estimated cost of an item in one year, given the current price of the item and the inflation rate as arguments.

5 Write a function declaration for a function that computes interest on a credit card account balance. The function takes arguments for the initial balance, the monthly interest rate, and the number of months for which interest must be paid. The value returned is the interest due. Do not forget to compound the interest—that is, to charge interest on the interest due. The interest due is added into the balance due, and the interest for the next month is computed using this larger balance. Use a *while* loop that is similar to (but need not be identical to) the one shown in Display 2.14. Embed the function in a program that reads the values for the interest rate, initial account balance, and number of months, then outputs the interest due. Embed your function definition in a program that lets the user compute interest due on a credit account balance. The program should allow the user to repeat the calculation until the user said he or she wants to end the program.

6 The gravitational attractive force between two bodies with masses m_1 and m_2 separated by a distance d is given by:

$$F = \frac{Gm_1m_2}{d^2}$$

where G is the universal gravitational constant:

$G = 6.673 \times 10^{-8}\ \text{cm}^3/(\text{g} \bullet \text{sec}^2)$

Write a function definition that takes arguments for the masses of two bodies and the distance between them, and that returns the gravitational force between them. Since you will use the above formula, the gravitational force will be in dynes. One dyne equals a

$\text{g} \bullet \text{cm/sec}^2$

You should use a globally defined constant for the universal gravitational constant. Embed your function definition in a complete program that computes the gravitational force between two objects given suitable inputs. Your program should allow the user to repeat this calculation as often as the user wishes.

7 Write a program that computes the annual after-tax cost of a new house for the first year of ownership. The cost is computed as the annual mortgage cost minus the tax savings. The input should be the price of the house and the down payment. The annual mortgage cost can be estimated as 3% of the initial loan balance credited toward paying off the loan principal plus 8% of the initial loan balance in interest. The initial loan balance is the price minus the down payment. Assume a 35% marginal tax rate and assume that interest payments are tax deductible. So, the tax savings is 35% of the interest payment. Your program should use at least two function definitions. Your program should allow the user to repeat this calculation as often as the user wishes.

8 Write a program that asks for the user's height, weight, and age, and then computes clothing sizes according to the formulas:

- Hat size = weight in pounds divided by height in inches and all that multiplied by 2.9.
- Jacket size (chest in inches) = height times weight divided by 288 and then adjusted by adding 1/8 of an inch for each 10 years over age 30. (Note that

the adjustment only takes place after a full 10 years. So, there is no adjustment for ages 30 through 39, but 1/8 of an inch is added for age 40.)

- Waist in inches = weight divided by 5.7 and then adjusted by adding 1/10 of an inch for each 2 years over age 28. (Note that the adjustment only takes place after a full 2 years. So, there is no adjustment for age 29, but 1/10 of an inch is added for age 30.)

Use functions for each calculation. Your program should allow the user to repeat this calculation as often as the user wishes.

9 That we are "blessed" with several absolute value functions is an accident of history. C libraries were already available when C++ arrived; they could be easily used, so they were not rewritten using function overloading. You are to find all the absolute value functions you can, and rewrite all of them overloading the `abs` function name. At a minimum you should have the *`int`*, *`long`*, *`float`*, and *`double`* types represented.

For additional online Programming Projects, click the CodeMate icons below.

3.11