Part IV

Structures and File Input/Output

Structures

Using structures, you have the ability to group data and work with the grouped data as a whole. Business data processing uses the concept of structures in almost every program. Being able to manipulate several variables as a single group makes your programs easier to manage.

This chapter introduces the following concepts:

- Structure definitions
- Initializing structures
- The dot operator (.)
- Structure assignment
- Nested structures

This chapter is one of the last in the book to present new concepts. The remainder of the book builds on the structure concepts you learn in this chapter.

Introduction to Structures

Structures can have members of different data types.

A *structure* is a collection of one or more variable types. As you know, each element in an array must be the same data type, and you must refer to the entire array by its name. Each element (called a *member*) in a structure can be a different data type.

Suppose you wanted to keep track of your CD music collection. You might want to track the following pieces of information about each CD:

Title
Artist
Number of songs
Cost
Date purchased

There would be five members in this CD structure.

TIP: If you have programmed in other computer languages, or if you have ever used a database program, C++ structures are analogous to file records, and members are analogous to fields in those records.

After deciding on the members, you must decide what data type each member is. The title and artist are character arrays, the number of songs is an integer, the cost is floating-point, and the date is another character array. This information is represented like this:

Member Name	*Data Type*
Title	Character array of 25 characters
Artist	Character array of 20 characters
Number of songs	Integer
Cost	Floating-point
Date purchased	Character array of eight characters

Each structure you define can have an associated structure name called a *structure tag.* Structure tags are not required in most cases, but it is generally best to define one for each structure in your program. The structure tag is not a variable name. Unlike array names, which reference the array as variables, a structure tag is simply a label for the structure's format.

You name structure tags yourself, using the same naming rules for variables. If you give the CD structure a structure tag named `cd_collection`, you are informing C++ that the tag called `cd_collection` looks like two character arrays, followed by an integer, a floating-point value, and a final character array.

A structure tag is a label for the structure's format.

A structure tag is actually a newly defined data type that you, the programmer, define. When you want to store an integer, you do not have to define to C++ what an integer is. C++ already recognizes an integer. When you want to store a CD collection's data, however, C++ is not capable of recognizing what format your CD collection takes. You have to tell C++ (using the example being described here) that you need a new data type. That data type will be your structure tag, called `cd_collection` in this example, and it looks like the structure previously described (two character arrays, integer, floating-point, and character array).

NOTE: No memory is reserved for structure tags. A structure tag is your own data type. C++ does not reserve memory for the integer data type until you declare an integer variable. C++ does not reserve memory for a structure until you declare a structure variable.

Figure 28.1 shows the CD structure, graphically representing the data types in the structure. Notice that there are five members and each member is a different data type. The entire structure is called `cd_collection` because that is the structure tag.

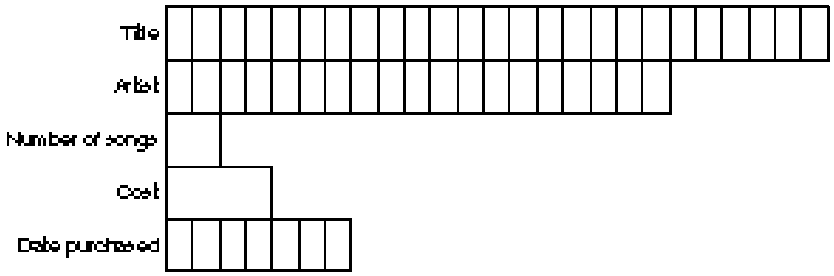
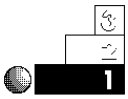


Figure 28.1. The layout of the `cd_collection` structure.

NOTE: The mailing-list application in Appendix F uses a structure to hold people's names, addresses, cities, states, and ZIP codes.

Examples



1. Suppose you were asked to write a program for a company's inventory system. The company had been using a card-file inventory system to track the following items:

   Item name
   Quantity in stock
   Quantity on order
   Retail price
   Wholesale price

   This would be a perfect use for a structure containing five members. Before defining the structure, you have to determine the data types of each member. After asking questions about the range of data (you must know the largest item name, and the highest possible quantity that would appear on order to ensure your data types can hold the data), you decide to use the following structure tag and data types:

Member	*Data Type*
Item name	Character array of 20 characters
Quantity in stock	`long int`
Quantity on order	`long int`
Retail price	`double`
Wholesale price	`double`

2. Suppose the same company also wanted you to write a program to keep track of their monthly and annual salaries and to print a report at the end of the year that showed each month's individual salary and the total salary at the end of the year.

   What would the structure look like? Be careful! This type of data probably does not need a structure. Because all the monthly salaries must be the same data type, a floating-point or a double floating-point array holds the monthly salaries nicely without the complexity of a structure.

   Structures are useful for keeping track of data that must be grouped, such as inventory data, a customer's name and address data, or an employee data file.

Defining Structures

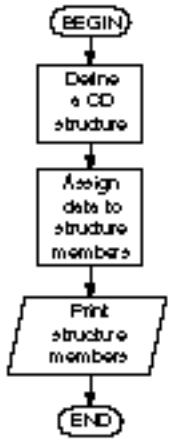
To define a structure, you must use the `struct` statement. The `struct` statement defines a new data type, with more than one member, for your program. The format of the `struct` statement is

```
struct [structure tag]
   {
      member definition;
      member definition;
        :
      member definition;
   } [one or more structure variables];
```

As mentioned earlier, `structure tag` is optional (hence the brackets in the format). Each `member definition` is a normal variable definition, such as `int i;` or `float sales[20];` or any other valid variable definition, including variable pointers if the structure requires a pointer as a member. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables.

If you specify a structure variable, you request C++ to reserve space for that variable. This enables C++ to recognize that the variable is not integer, character, or any other internal data type. C++ also recognizes that the variable must be a type that looks like the structure. It might seem strange that the members do not reserve storage, but they don't. The structure variables do. This becomes clear in the examples that follow.

Here is the way you declare the CD structure:




```
struct cd_collection
   {
     char title[25];
     char artist[20];
     int num_songs;
     float price;
     char date_purch[9];
   } cd1, cd2, cd3;
```

Before going any further, you should be able to answer the following questions about this structure:

- What is the structure tag?
- How many members are there?
- What are the member data types?
- What are the member names?
- How many structure variables are there?
- What are their names?

The structure tag is called `cd_collection`. There are five members, two character arrays, an integer, a floating-point, and a character array. The member names are `title`, `artist`, `num_songs`, `price`, and `date_purch`. There are three structure variables—`cd1`, `cd2`, and `cd3`.

TIP: Often, you can visualize structure variables as a card-file inventory system. Figure 28.2 shows how you might keep your CD collection in a 3-by-5 card file. Each CD takes one card (represented by its structure variable), which contains the information about that CD (the structure members).

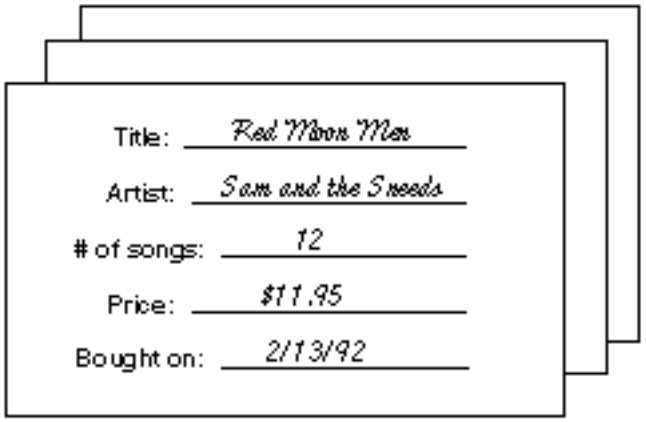
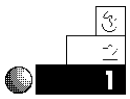


Figure 28.2. Using a card-file CD inventory system.

If you had 1000 CDs, you would have to declare 1000 structure variables. Obviously, you would not want to list that many structure variables at the end of a structure definition. To help define structures for a large number of occurrences, you must define an *array of structures*. Chapter 29, "Arrays of Structures," shows you how to do that. For now, concentrate on familiarizing yourself with structure definitions.

Examples



1. Here is a structure definition of the inventory application described earlier in this chapter.

```
struct inventory
{
   char item_name[20];
   long int in_stock;
   long int order_qty;
   float retail;
   float wholesale;
} item1, item2, item3, item4;
```

Four inventory structure variables are defined. Each structure variable—`item1`, `item2`, `item3`, and `item4`—looks like the structure.

2. Suppose a company wanted to track its customers and personnel. The following two structure definitions would create five structure variables for each structure. This example, having five employees and five customers, is very limited, but it shows how structures can be defined.

```
struct employees
{
   char emp_name[25];              // Employee's full name.
   char address[30];                 // Employee's address.
   char city[10];
   char state[2];
   long int zip;
   double salary;                          // Annual salary.
} emp1, emp2, emp3, emp4, emp5;

struct customers
{
   char cust_name[25];             // Customer's full name.
   char address[30];                 // Customer's address.
   char city[10];
   char state[2];
   long int zip;
   double balance;              // Balance owed to company.
} cust1, cust2, cust3, cust4, cust5;
```

Each structure has similar data. Later in this chapter, you learn how to consolidate similar member definitions by creating nested structures.

TIP: Put comments to the right of members in order to document the purpose of the members.

Initializing Structure Data

You can define a structure's data when you declare the structure.

There are two ways to initialize members of a structure. You can initialize members when you declare a structure, and you can initialize a structure in the body of the program. Most programs lend themselves to the latter method, because you do not always know structure data when you write your program.

Here is an example of a structure declared and initialized at the same time:

```
struct cd_collection
   {
     char title[25];
     char artist[20];
     int num_songs;
     float price;
     char date_purch[9];
   } cd1 = {"Red Moon Men", "Sam and the Sneeds",
            12, 11.95, "02/13/92"};
```

When first learning about structures, you might be tempted to initialize members individually inside the structure, such as

```
char artist[20]="Sam and the Sneeds";       // Invalid
```

You cannot initialize individual members because they are not variables. You can assign only values to variables. The only structure variable in this structure is `cd1`. The braces must enclose the data you initialize in the structure variables, just as they enclose data when you initialize arrays.

This method of initializing structure variables becomes tedious when there are several structure variables (as there usually are). Putting the data in several variables, each set of data enclosed in braces, becomes messy and takes too much space in your code.

More importantly, you usually do not even know the contents of the structure variables. Generally, the user enters data to be stored in structures, or you read them from a disk file.

Use the dot operator to initialize members of structures.

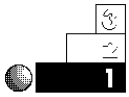
A better approach to initializing structures is to use the *dot operator* (.). The dot operator is one way to initialize individual members of a structure variable in the body of your program. With the dot operator, you can treat each structure member almost as if it were a regular nonstructure variable.

The format of the dot operator is

```
structure_variable_name.member_name
```

A structure variable name must always precede the dot operator, and a member name must always appear after the dot operator. Using the dot operator is easy, as the following examples show.

Examples



1. Here is a simple program using the CD collection structure and the dot operator to initialize the structure. Notice the program treats members as if they were regular variables when combined with the dot operator.

Identify the program and include the necessary header file. Define a CD structure variable with five members. Fill the CD structure variable with data, then print it.

```
// Filename: C28ST1.CPP
// Structure initialization with the CD collection.
#include <iostream.h>
#include <string.h>
void main()
{
   struct cd_collection
   {
     char title[25];
     char artist[20];
     int num_songs;
     float price;
     char date_purch[9];
   } cd1;
```

```
    // Initialize members here.
    strcpy(cd1.title, "Red Moon Men");
    strcpy(cd1.artist, "Sam and the Sneeds");
    cd1.num_songs=12;
    cd1.price=11.95;
    strcpy(cd1.date_purch, "02/13/92");

    // Print the data to the screen.
    cout << "Here is the CD information:\n\n";
    cout << "Title: " << cd1.title << "\n";
    cout << "Artist: " << cd1.artist << "\n";
    cout << "Songs: " << cd1.num_songs << "\n";
    cout << "Price: " << cd1.price << "\n";
    cout << "Date purchased: " << cd1.date_purch << "\n";

    return;
}
```

Here is the output from this program:

```
Here is the CD information:

Title: Red Moon Men
Artist: Sam and the Sneeds
Songs: 12
Price: 11.95
Date purchased: 02/13/92
```

2. By using the dot operator, you can receive structure data from the keyboard with any of the data-input functions you know, such as `cin`, `gets()`, and `get`.

   The following program asks the user for student information. To keep the example reasonably short, only two students are defined in the program.

```
// Filename: C28ST2.CPP
// Structure input with student data.
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
#include <stdio.h>
```

```
void main()
{
   struct students
   {
     char name[25];
     int age;
     float average;
   } student1, student2;

   // Get data for two students.
   cout << "What is first student's name? ";
   gets(student1.name);
   cout << "What is the first student's age? ";
   cin >> student1.age;
   cout << "What is the first student's average? ";
   cin >> student1.average;

   fflush(stdin);    // Clear input buffer for next input.

   cout << "\nWhat is second student's name? ";
   gets(student2.name);
   cout << "What is the second student's age? ";
   cin >> student2.age;
   cout << "What is the second student's average? ";
   cin >> student2.average;

   // Print the data.
   cout << "\n\nHere is the student information you " <<
           "entered:\n\n";
   cout << "Student #1:\n";
   cout << "Name:    " << student1.name << "\n";
   cout << "Age:     " << student1.age << "\n";
   cout << "Average: " << setprecision(2) << student1.average
                       << "\n";

   cout << "\nStudent #2:\n";
   cout << "Name:    " << student2.name << "\n";
   cout << "Age:     " << student2.age << "\n";
   cout << "Average: " << student2.average << "\n";

   return;
}
```

Here is the output from this program:

```
What is first student's name? Larry
What is the first student's age? 14
What is the first student's average? 87.67

What is second student's name? Judy
What is the second student's age? 15
What is the second student's average? 95.38


Here is the student information you entered:

Student #1:
Name:    Larry
Age:     14
Average: 87.67

Student #2:
Name:    Judy
Age:     15
Average: 95.38
```

3. Structure variables are passed by copy, not by address as arrays are. Therefore, if you fill a structure in a function, you must return it to the calling function in order for the calling function to recognize the structure, or use global structure variables, which is generally not recommended.

TIP: A good solution to the local/global structure problem is this: Define your structures globally without any structure variables. Define all your structure variables locally to the functions that need them. As long as your structure definition is global, you can declare local structure variables from that structure. All subsequent examples in this book use this method.

Define structures globally and structure variables locally.

The structure tag plays an important role in the local/global problem. Use the structure tag to define local structure variables. The following program is similar to the previous one. Notice the student structure is defined globally with no

structure variables. In each function, local structure variables are declared by referring to the structure tag. The structure tag keeps you from having to redefine the structure members every time you define a new structure variable.

```
// Filename: C28ST3.CPP
// Structure input with student data passed to functions.
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <iomanip.h>
struct students fill_structs(struct students student_var);
void pr_students(struct students student_var);

struct students                      // A global structure.
   {
     char name[25];
     int age;
     float average;
   };                             // No memory reserved.

void main()
{
   students student1, student2;      // Defines two
                                     // local variables.

   // Call function to fill structure variables.
   student1 = fill_structs(student1);         // student1
                 // is passed by copy, so it must be
                 // returned for main() to recognize it.
   student2 = fill_structs(student2);

   // Print the data.
   cout << "\n\nHere is the student information you";
   cout << " entered:\n\n";
   pr_students(student1);   // Prints first student's data.
   pr_students(student2);  // Prints second student's data.

   return;
}
```

```
struct students fill_structs(struct students student_var)
{
   // Get student's data
   fflush(stdin);    // Clears input buffer for next input.
   cout << "What is student's name? ";
   gets(student_var.name);
   cout << "What is the student's age? ";
   cin >> student_var.age;
   cout << "What is the student's average? ";
   cin >> student_var.average;

   return (student_var);
}

void pr_students(struct students student_var)
{
   cout << "Name:    " << student_var.name << "\n";
   cout << "Age:     " << student_var.age << "\n";
   cout << "Average: " << setprecision(2) <<
                          student_var.average << "\n";
   return;
}
```

The prototype and definition of the `fill_structs()` function might seem complicated, but it follows the same pattern you have seen throughout this book. Before a function name, you must declare `void` or put the return data type if the function returns a value. `fill_structs()` does return a value, and the type of value it returns is `struct students`.

4. Because structure data is nothing more than regular variables grouped together, feel free to calculate using structure members. As long as you use the dot operator, you can treat structure members just as you would other variables.

   The following example asks for a customer's balance and uses a discount rate, included in the customer's structure, to calculate a new balance. To keep the example short, the structure's data is initialized at variable declaration time.

   This program does not actually require structures because only one customer is used. Individual variables could have

been used, but they don't illustrate the concept of calculating with structures.

```
// Filename: C28CUST.CPP
// Updates a customer balance in a structure.
#include <iostream.h>
#include <iomanip.h>

struct customer_rec
   {
      char cust_name[25];
      double balance;
      float dis_rate;
   } ;

void main()
{
   struct customer_rec customer = {"Steve Thompson",
                                    431.23, .25};

   cout << "Before the update, " << customer.cust_name;
   cout << " has a balance of $" << setprecision(2) <<
           customer.balance << "\n";

   // Update the balance
   customer.balance *= (1.0-customer.dis_rate);

   cout << "After the update, " << customer.cust_name;
   cout << " has a balance of $" << customer.balance << "\n";
   return;
}
```

5. You can copy the members of one structure variable to the members of another as long as both structures have the same format. Some older versions of C++ require you to copy each member individually when you want to copy one structure variable to another, but AT&T C++ makes duplicating structure variables easy.

EXAMPLE

Being able to copy one structure variable to another will seem more meaningful when you read Chapter 29, “Arrays of Structures.”

The following program declares three structure variables, but initializes only the first one with data. The other two are then initialized by assigning the first structure variable to them.


```
// Filename: C28STCPY.CPP
// Demonstrates assigning one structure to another.
#include <iostream.h>
#include <iomanip.h>

struct student
{
   char st_name[25];
   char grade;
   int age;
   float average;
};

void main()
{
   student std1 = {“Joe Brown”, ‘A’, 13, 91.4};
   struct student std2, std3;          // Not initialized

   std2 = std1;             // Copies each member of std1
   std3 = std1;             // to std2 and std3.


   cout << “The contents of std2:\n”;
   cout << std2.st_name << “ “ << std2.grade << “ “;
   cout << std2.age << “ “ << setprecision(1) << std2.average
        << “\n\n”;

   cout << “The contents of std3:\n”;
   cout << std3.st_name << “ “ << std3.grade << “ “;
   cout << std3.age << “ “ << std3.average << “\n”;
   return;
}
```

Here is the output from the program:

```
The contents of std2
Joe Brown, A, 13, 91.4

The contents of std3
Joe Brown, A, 13, 91.4
```

Notice each member of `std1` was assigned to `std2` and `std3` with two single assignments.

Nested Structures

C++ gives you the ability to nest one structure definition in another. This saves time when you are writing programs that use similar structures. You have to define the common members only once in their own structure and then use that structure as a member in another structure.

The following two structure definitions illustrate this point:

```
struct employees
{
   char emp_name[25];          // Employee's full name.
   char address[30];           // Employee's address.
   char city[10];
   char state[2];
   long int zip;
   double salary;              // Annual salary.
};

struct customers
{
   char cust_name[25];         // Customer's full name.
   char address[30];           // Customer's address.
   char city[10];
   char state[2];
   long int zip;
   double balance;             // Balance owed to company.
};
```

These structures hold different data. One structure is for employee data and the other holds customer data. Even though the data should be kept separate (you don't want to send a customer a paycheck!), the structure definitions have much overlap and can be consolidated by creating a third structure.

Suppose you created the following structure:

```
struct address_info
{
   char address[30];        // Common address information.
   char city[10];
   char state[2];
   long int zip;
};
```

This structure could then be used as a member in the other structures like this:

```
struct employees
{
   char emp_name[25];         // Employee's full name.
   address_info e_address;    // Employee's address.
   double salary;             // Annual salary.
};

struct customers
{
   char cust_name[25];        // Customer's full name.
   address_info c_address;    // Customer's address.
   double balance;            // Balance owed to company.
};
```

It is important to realize there are a total of three structures, and that they have the tags `address_info`, `employees`, and `customers`. How many members does the `employees` structure have? If you answered three, you are correct. There are three members in both `employees` and `customers`. The `employees` structure has the structure of a character array, followed by the `address_info` structure, followed by the double floating-point member, `salary`.

Figure 28.3 shows how these structures look.

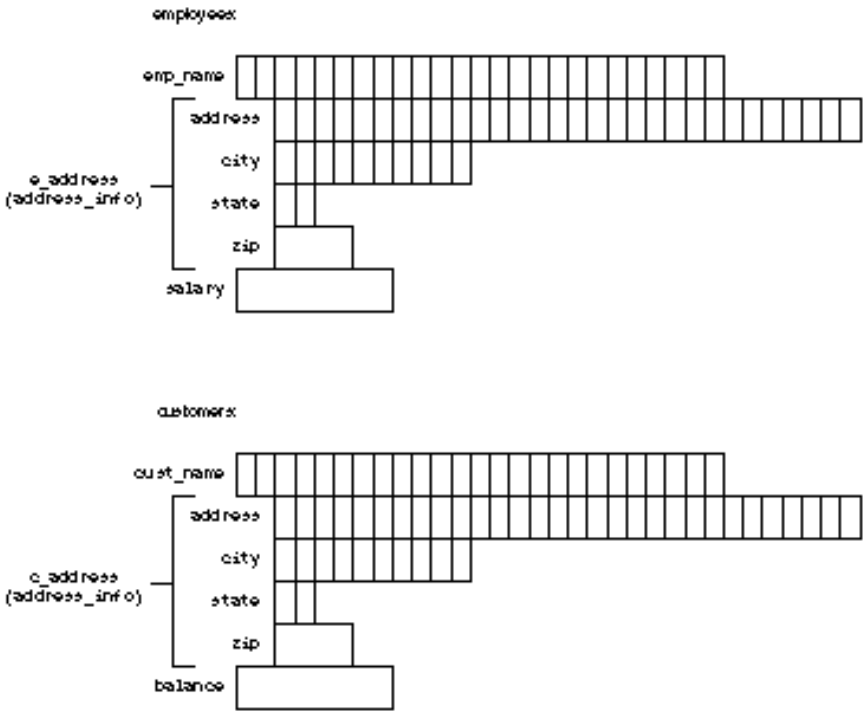


Figure 28.3. Defining a nested structure.

When you define a structure, that structure becomes a new data type in the program and can be used anywhere a data type (such as `int`, `float`, and so on) can appear.

You can assign members values using the dot operator. To assign the customer balance a number, type something like this:

```
customer.balance = 5643.24;
```

The nested structure might seem to pose a problem. How can you assign a value to one of the nested members? By using the dot operator, you must nest the dot operator just as you nest the structure definitions. You would assign a value to the customer's ZIP code like this:

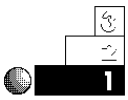
```
customer.c_address.zip = 34312;
```

To assign a value to the employee's ZIP code, you would do this:

```
employee.e_address.zip = 59823;
```

Review Questions

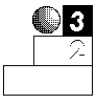
The answers to the review questions are in Appendix B.



1. What is the difference between structures and arrays?
2. What are the individual elements of a structure called?
3. What are the two ways to initialize members of a structure?

4. Do you pass structures by copy or by address?
5. True or false: The following structure definition reserves storage in memory:

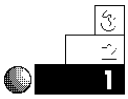
```
struct crec
   { char name[25];
     int age;
     float sales[5];
     long int num;
   }
```



6. Should you declare a structure globally or locally?
7. Should you declare a structure variable globally or locally?
8. How many members does the following structure declaration contain?

```
struct item
   {
     int quantity;
     part_rec item_desc;
     float price;
     char date_purch[8];
   };
```

Review Exercises



1. Write a structure in a program that tracks a video store's tape inventory. Be sure the structure includes the tape title, the length of the tape (in minutes), the initial purchase price of the tape, the rental price of the tape, and the date of the movie's release.

2. Write a program using the structure you declared in Exercise 1. Define three structure variables and initialize them when you declare the variables with data. Print the data to the screen.

3. Write a teacher's program to keep track of 10 students' names, ages, letter grades, and IQs. Use 10 different structure variable names and retrieve the data for the students in a `for` loop from the keyboard. Print the data on the printer when the teacher finishes entering the information for all the students.

Summary

With structures, you have the ability to group data in more flexible ways than with arrays. Your structures can contain members of different data types. You can initialize the structures either at declaration time or during the program with the dot operator.

Structures become even more powerful when you declare arrays of structure variables. Chapter 29, "Arrays of Structures," shows you how to declare several structure variables without giving them each a different name. This enables you to step through structures much quicker with loop constructs.

Variable Scope

The concept of *variable scope* is most important when you write functions. Variable scope determines which functions recognize certain variables. If a function recognizes a variable, the variable is *visible* to that function. Variable scope protects variables in one function from other functions that might overwrite them. If a function doesn't need access to a variable, that function shouldn't be able to see or change the variable. In other words, the variable should not be "visible" to that particular function.

This chapter introduces you to

- Global and local variables
- Passing arguments
- Automatic and static variables
- Passing parameters

The previous chapter introduced the concept of using a different function for each task. This concept is much more useful when you learn about local and global variable scope.

Global Versus Local Variables

If you have programmed only in BASIC, the concept of local and global variables might be new to you. In many interpreted versions of BASIC, all variables are *global*, meaning the entire program knows each variable and has the capability to change any of them. If you use a variable called SALES at the top of the program, even the last line in the program can use SALES. (If you don't know BASIC, don't despair—there will be one less habit you have to break!)

Global variables can be dangerous. Parts of a program can inadvertently change a variable that shouldn't be changed. For example, suppose you are writing a program that keeps track of a grocery store's inventory. You might keep track of sales percentages, discounts, retail prices, wholesale prices, produce prices, dairy prices, delivered prices, price changes, sales tax percentages, holiday markups, post-holiday markdowns, and so on.

The huge number of prices in such a system is confusing. When writing a program to keep track of every price, it would be easy to mistakenly call both the dairy prices `d_prices` and the delivered prices `d_prices`. Either C++ will not enable you to do this (you can't define the same variable twice) or you will overwrite a value used for something else. Whatever happens, keeping track of all these different—but similarly named—prices makes this program confusing to write.

Global variables are visible across many program functions.

Global variables can be dangerous because code can inadvertently overwrite a variable initialized elsewhere in the program. It is better to make every variable *local* in your programs. Then, only functions that should be able to change the variables can do so.

Local variables are visible only in the block where they are defined.

Local variables can be seen (and changed) only from the function in which they are defined. Therefore, if a function defines a variable as local, that variable's scope is protected. The variable cannot be used, changed, or erased by any other function without special programming that you learn about shortly.

If you use only one function, `main()`, the concept of local and global is academic. You know from Chapter 16, "Writing C++ Functions," however, that single-function programs are not recommended. It is best to write modular, structured programs made up

of many smaller functions. Therefore, you should know how to define variables as local to only those functions that use them.

Defining Variable Scope

When you first learned about variables in Chapter 4, "Variables and Literals," you learned you can define variables in two places:

- Before they are used inside a function
- Before a function name, such as `main()`

All examples in this book have declared variables with the first method. You have yet to see an example of the second method. Because most these programs have consisted entirely of a single `main()` function, there has been no reason to differentiate the two methods. It is only after you start using several functions in one program that these two variable definition methods become critical.

The following rules, specific to local and global variables, are important:

- A variable is local *if and only if* you define it after the opening brace of a block, usually at the top of a function.
- A variable is global *if and only if* you define it outside a function.

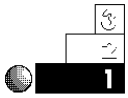
All variables you have seen so far have been local. They have all been defined immediately after the opening braces of `main()`. Therefore, they have been local to `main()`, and only `main()` can use them. Other functions have no idea these variables even exist because they belong to `main()` only. When the function (or block) ends, all its local variables are destroyed.

TIP: All local variables disappear (lose their definition) when their block ends.

Global variables are visible from their definition through the remainder of the program.

Global variables are visible ("known") from their point of definition to the end of the program. If you define a global variable, *any* line throughout the rest of the program—no matter how many functions and code lines follow it—is able to use that global variable.

Examples



1. The following section of code defines two local variables, `i` and `j`.

```
main()
{
   int i, j;                    // Local because they're
                                // defined after the brace.
   // Rest of main() goes here.
}
```

   These variables are visible to `main()` and not to any other function that might follow or be called by `main()`.

2. The following section of code defines two global variables, `g` and `h`.

```
#include <iostream.h>
int g, h;                    // Global because they're
                             // defined before a function.
main()
{
   // main()'s code goes here.
}
```

   It doesn't matter whether your `#include` lines go before or after global variable declarations.

3. Global variables can appear before any function. In the following program, `main()` uses no variables. However, both of the two functions after `main()` can use `sales` and `profit` because these variables are global.

```
// Filename: C17GLO.CPP
// Program that contains two global variables.
#include <iostream.h>
do_fun();
third_fun();  // Prototype discussed later.
main()
{
   cout << "No variables defined in main() \n\n";
   do_fun();                 // Call the first function.
```

```
   return 0;
}

float sales, profit;              // Two global variables.
do_fun()
{
   sales = 20000.00;         // This variable is visible
                             // from this point down.
   profit = 5000.00;         // As is this one. They are
                             // both global.

   cout << "The sales in the second function are " <<
           sales << "\n";
   cout << "The profit in the second function is " <<
           profit << "\n\n";

   third_fun();           // Call the third function to
                          // show that globals are visible.
   return 0;
}

third_fun()
{
   cout << "In the third function: \n";
   cout << "The sales in the third function are " <<
           sales << "\n";
   cout << "The profit in the third function is " <<
           profit << "\n";
   // If sales and profit were local, they would not be
   // visible by more than one function.
   return 0;
}
```

Notice that the `main()` function can never use `sales` and `profit` because they are not visible to `main()`—even though they are global. Remember, global variables are visible only from their point of definition downward in the program. Statements that appear before global variable definitions cannot use those variables. Here is the result of running this program.

```
No variables defined in main()

The sales in the second function are 20000
The profit in the second function is 5000

In the third function:
The sales in the third function are 20000
The profit in the third function is 5000
```

TIP: Declare all global variables at the top of your programs. Even though you can define them later (between any two functions), you can find them faster if you declare them at the top.

4. The following program uses both local and global variables. It should now be obvious to you that `j` and `p` are local and `i` and `z` are global.

```
// Filename: C17GLLO.CPP
// Program with both local and global variables.
// Local Variables       Global Variables
//     j, p                    i, z
#include <iostream.h>
pr_again();  // Prototype

int i = 0;                // Global variable because it's
                                // defined outside main().
main()
{
   float p ;                        // Local to main() only.
   p = 9.0;                // Puts value in global variable.
   cout << i << ", " << p << "\n";      // Prints global i
                                          //  and local p.
   pr_again();                     // Calls next function.
   return 0;                            // Returns to DOS.

 }
```

```
float z = 9.0;              // Global variable because it's
                              //  defined before a function.
pr_again()
{
   int j = 5;                   // Local to only pr_again().
   cout << j << ", " << z;       // This can't print p!.
   cout << ", " << i << "\n";
   return 0;                              // Return to main().
 }
```

Even though `j` is defined in a function that `main()` calls, `main()` cannot use `j` because `j` is local to `pr_again()`. When `pr_again()` finishes, `j` is no longer defined. The variable `z` is global from its point of definition down. This is why `main()` cannot print `z`. Also, the function `pr_again()` cannot print `p` because `p` is local to `main()` only.

Make sure you can recognize local and global variables before you continue. A little study here makes the rest of this chapter easy to understand.

5. Two variables can have the same name, as long as they are local to two different functions. They are distinct variables, even though they are named identically.

   The following short program uses two variables, both named `age`. They have two different values, and they are considered to be two different variables. The first `age` is local to `main()`, and the second `age` is local to `get_age()`.

```
// Filename: C17LOC1.CPP
// Two different local variables with the same name.
#include <iostream.h>
get_age();  // Prototype
main()
{
   int age;
   cout << "What is your age? ";
   cin >> age;

   get_age();                  // Call the second function.
   cout << "main()'s age is still " << age << "\n";
```

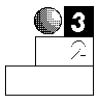
```
   return 0;
}

get_age()
{
   int age;                  // A different age. This one
                                //  is local to get_age().
   cout << "What is your age again? ";
   cin >> age;
   return 0;
}
```

Variables local to `main()` cannot be used in another function that `main()` calls.

The output of this program follows. Study this output carefully. Notice that `main()`'s last `cout` does not print the newly changed `age`. Rather, it prints the `age` known to `main()`—the `age` that is *local* to `main()`. Even though they are named the same, `main()`'s `age` has nothing to do with `get_age()`'s `age`. They might as well have two different variable names.

```
What is your age? 28
What is your age again? 56
main()'s age is still 28
```

You should be careful when naming variables. Having two variables with the same name is misleading. It would be easy to become confused while changing this program later. If these variables truly have to be separate, name them differently, such as `old_age` and `new_age`, or `ag1` and `ag2`. This helps you remember that they are different.



6. There are a few times when overlapping local variable names does not add confusion, but be careful about overdoing it. Programmers often use the same variable name as the counter variable in a `for` loop. For example, the two local variables in the following program have the same name.

```
// Filename: C17LOC2.CPP
// Using two local variables with the same name
```

```
// as counting variables.
#include <iostream.h>
do_fun();  // Prototype
main()
{
   int ctr;                                  // Loop counter.
   for (ctr=0; ctr<=10; ctr++)
      { cout << "main()'s ctr is " << ctr << "\n"; }
   do_fun();                        // Call second function.

   return 0;
}

do_fun()
{
   int ctr;
   for (ctr=10; ctr>=0; ctr--)
      { cout << "do_fun()'s ctr is " << ctr << "\n"; }
   return 0;                                 // Return to main().
}
```

Although this is a nonsense program that simply prints `0` through `10` and then prints `10` through `0`, it shows that using `ctr` for both function names is not a problem. These variables do not hold important data that must be processed; rather, they are `for` loop-counting variables. Calling them both `ctr` leads to little confusion because their use is limited to controlling `for` loops. Because a `for` loop initializes and increments variables, the one function never relies on the other function's `ctr` to do anything.

7. Be careful about creating local variables with the same name in the same function. If you define a local variable early in a function and then define another local variable with the same name inside a new block, C++ uses only the innermost variable, until its block ends.

   The following example helps clarify this confusing problem. The program contains one function with three local variables. See if you can find these three variables.

```
// Filename: C17MULI.CPP
// Program with multiple local variables called i.
#include <iostream.h>
main()
{
   int i;                                  // Outer i
   i = 10;

   { int i;                                // New block's i
     i = 20;                    // Outer i still holds a 10.
     cout << i << " " << i  << "\n";       // Prints 20 20.

     { int i;     // Another new block and local variable.
       i = 30;                          // Innermost i only.
       cout << i << " " << i <<
               " " << i << "\n";         // Prints 30 30 30.
     }                 // Innermost i is now gone forever.

    }       // Second i is gone forever (its block ended).

   cout << i << " " << i << " " <<
           i << "\n";                    // Prints 10 10 10.
   return 0;
}                  // main() ends and so do its variables.
```

All local variables are local to the block in which they are defined. This program has three blocks, each one nested within another. Because you can define local variables immediately after an opening brace of a block, there are three distinct `i` variables in this program.

The local `i` disappears completely when its block ends (when the closing brace is reached). C++ always prints the variable that it interprets as the most local—the one that resides within the innermost block.

Use Global Variables Sparingly

You might be asking yourself, "Why do I have to understand global and local variables?" At this point, that is an understandable

question, especially if you have been programming mostly in BASIC. Here is the bottom line: Global variables can be *dangerous.* Code can inadvertently overwrite a variable that was initialized in another place in the program. It is better to have every variable in your program be *local to the function that has to access it.*

Read the last sentence again. Even though you now know how to make variables global, you should avoid doing so! Try to never use another global variable. It might seem easier to use global variables when you write programs having more than one function: If you make every variable used by every function global, you never have to worry whether one is visible or not to any given function. On the other hand, a function can accidentally change a global variable when that was not your intention. If you keep variables local only to functions that need them, you protect their values, and you also keep your programs fully modular.

The Need for Passing Variables

You just learned the difference between local and global variables. You saw that by making your variables local, you protect their values because the function that sees the variable is the only one that can modify it.

What do you do, however, if you have a local variable you want to use in *two or more* functions? In other words, you might need a variable to be both added from the keyboard in one function and printed in another function. If the variable is local only to the first function, how can the second one access it?

You have two solutions if more than one function has to share a variable. One, you can declare the variable globally. This is not a good idea because you want only those two functions to have access to the variable, but all functions have access to it when it's global. The other alternative—and the better one by far—is to *pass* the local variable from one function to another. This has a big advantage: The variable is only known to those two functions. The rest of the program still has no access to it.

CAUTION: Never pass a global variable to a function. There is no reason to pass global variables anyway because they are already visible to all functions.

You pass an argument when you pass one local variable to another function.

When you pass a local variable from one function to another, you *pass an argument* from the first function to the next. You can pass more than one argument (variable) at a time, if you want several local variables to be sent from one function to another. The receiving function *receives a parameter* (variable) from the function that sends it. You shouldn't worry too much about what you call them—either arguments or parameters. The important thing to remember is that you are sending local variables from one function to another.

NOTE: You have already passed arguments to parameters when you passed data to the `cout` operator. The literals, variables, and expressions in the `cout` parentheses are arguments. The built-in `cout` function receives these values (called parameters on the receiving end) and displays them.

A little more terminology is needed before you see some examples. When a function passes an argument, it is called the *calling function.* The function that receives the argument (called a parameter when it is received) is called the *receiving function.* Figure 17.1 explains these terms.

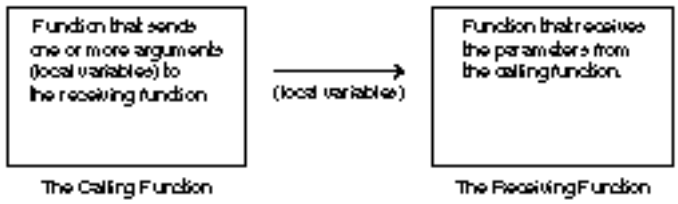


Figure 17.1. The calling and receiving functions.

If a function name has empty parentheses, nothing is being passed to it.

To pass a local variable from one function to another, you must place the local variable in parentheses in both the calling function and the receiving function. For example, the local and global

examples presented earlier did not pass local variables from `main()` to `do_fun()`. If a function name has empty parentheses, nothing is being passed to it. Given this, the following line passes two variables, `total` and `discount`, to a function called `do_fun()`.

```
do_fun(total, discount);
```

It is sometimes said that a variable or function is *defined*. This has nothing to do with the `#define` preprocessor directive, which defines literals. You define variables with statements such as the following:

```
int i, j;
int m=9;
float x;
char ara[] = "Tulsa";
```

These statements tell the program that you need these variables to be reserved. A function is defined when the C++ compiler reads the first statement in the function that describes the name and when it reads any variables that might have been passed to that function as well. Never follow a function definition with a semicolon, but always follow the statement that calls a function with a semicolon.

NOTE: To some C++ purists, a variable is only declared when you write `int i;` and only truly defined when you assign it a value, such as `i=7;`. They say that the variable is both declared and defined when you declare the variable and assign it a value at the same time, such as `int i=7;`.

The following program contains two function definitions, `main()` and `pr_it()`.

To practice passing a variable to a function, declare `i` *as an integer variable and make it equal to five. The passing (or calling) function is* `main()`*, and the receiving function is* `pr_it()`*. Pass the* `i` *variable to the* `pr_it()` *function, then go back to* `main()`*.*

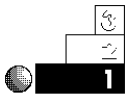
```
main()                    // The main() function definition.
{
  int i=5;                // Defines an integer variable.
  pr_it(i);               // Calls the pr_it().
                          // function and passes it i.
  return 0;               // Returns to the operating system.
}

pr_it(int i)              // The pr_it() function definition.
{
   cout << i << "\n";     // Calls the cout operator.
   return 0;              // Returns to main().
}
```

Because a passed parameter is treated like a local variable in the receiving function, the `cout` in `pr_it()` prints a `5`, even though the `main()` function initialized this variable.

When you pass arguments to a function, the receiving function is not aware of the data types of the incoming variables. Therefore, you must include each parameter's data type in front of the parameter's name. In the previous example, the definition of `pr_it()` (the first line of the function) contains the type, `int`, of the incoming variable `i`. Notice that the `main()` calling function does not have to indicate the variable type. In this example, `main()` already knows the type of variable `i` (an integer); only `pr_it()` has to know that `i` is an integer.

TIP: Always declare the parameter types in the receiving function. Precede each parameter in the function's parentheses with `int`, `float`, or whatever each passed variable's data type is.

Examples



1. Here is a `main()` function that contains three local variables. `main()` passes one of these variables to the first function and two of them to the second function.

```
// Filename: C17LOC3.CPP
// Pass three local variables to functions.
#include <iostream.h>
#include <iomanip.h>
pr_init(char initial);  // Prototypes discussed later.
pr_other(int age, float salary);

main()
{
   char initial;           // Three variables local to
                           // main().
   int age;
   float salary;

   // Fill these variables in main().
   cout << "What is your initial? ";
   cin >> initial;
   cout << "What is your age? ";
   cin >> age;
   cout << "What is your salary? ";
   cin >> salary;

   pr_init(initial);               // Call pr_init() and
                                      // pass it initial.
   pr_other(age, salary);          // Call pr_other() and
                                // pass it age and salary.

   return 0;
}

pr_init(char initial)         // Never put a semicolon in
                               // the function definition.
{
   cout << "Your initial is " << initial << "\n";
   return 0;                           // Return to main().
}

pr_other(int age, float salary) // Must type both parameters.
{
   cout << "You look young for " << age << "\n";
   cout << "And " << setprecision(2) << salary <<
```

```
            " is a LOT of money!";
   return 0;                              // Return to main().
}
```

2. A receiving function can contain its own local variables. As long as the names are not the same, these local variables do not conflict with the passed ones. In the following program, the second function receives a passed variable from `main()` and defines its own local variable called `price_per`.

```
// Filename: C17LOC4.CPP
// Second function has its own local variable.
#include <iostream.h>
#include <iomanip.h>
compute_sale(int gallons); // Prototypes discussed later.

main()
{
   int gallons;

   cout << "Richard's Paint Service \n";
   cout << "How many gallons of paint did you buy? ";
   cin >> gallons;           // Get gallons in main().

   compute_sale(gallons);    // Compute total in function.
   return 0;
}

compute_sale(int gallons)
{
   float price_per = 12.45;    // Local to compute_sale().

   cout << "The total is " << setprecision(2) <<
           (price_per*(float)gallons) << "\n";
         // Had to type cast gallons because it was integer.
   return 0;                              // Return to main().
}
```

3. The following sample code lines test your skill at recognizing calling functions and receiving functions. Being able to recognize the difference is half the battle of understanding them.

```
do_it()
```

The preceding fragment must be the first line of a new function because it does not end with a semicolon.

```
do_it2(sales);
```

This line calls a function called `do_it2()`. The calling function passes the variable called `sales` to `do_it2()`.

```
pr_it(float total)
```

The preceding line is the first line of a function that receives a floating-point variable from another function that called it. All receiving functions must specify the type of each variable being passed.

```
pr_them(float total, int number)
```

This is the first line of a function that receives two variables—one is a floating-point variable and the other is an integer. This line cannot be calling the function `pr_them` because there is no semicolon at the end of the line.

Automatic Versus Static Variables

The terms *automatic* and *static* describe what happens to local variables when a function returns to the calling procedure. By default, all local variables are automatic, meaning that they are erased when their function ends. You can designate a variable as automatic by prefixing its definition with the term `auto`. The `auto` keyword is optional with local variables because they are automatic be default.

The two statements after `main()`'s opening brace declare automatic local variables:

```
main()
{
   int i;
   auto float x;
   // Rest of main() goes here.
```

Because `auto` is the default, you did not have to include the term `auto` with `x`.

NOTE: C++ programmers rarely use the `auto` keyword with local variables because they are automatic by default.

Automatic variables are local and disappear when their function ends.

The opposite of an automatic variable is a static variable. All global variables are static and, as mentioned, all static variables retain their values. Therefore, if a local variable is static, it too retains its value when its function ends—in case the function is called a second time. To declare a variable as static, place the `static` keyword in front of the variable when you define it. The following code section defines three variables, `i`, `j`, and `k`. The variable `i` is automatic, but `j` and `k` are static.

```
my_fun()            // Start of new function definition.
{
   int i;
   static j=25;    // Both j and k are static variables.
   static k=30;
```

If local variables are static, their values remain in case the function is called again.

Always assign an initial value to a static variable when you declare it, as shown here in the last two lines. This initial value is placed in the static variable only the first time `my_fun()` executes. If you don't assign a static variable an initial value, C++ initializes it to zero.

TIP: Static variables are good to use when you write functions that keep track of a count or add to a total. If the counting or totaling variables were local and automatic, their values would disappear when the function finished—destroying the totals.

Automatic and Static Rules for Local Variables

Local automatic variables disappear when their block ends. All local variables are automatic by default. You can prefix a variable (when you define it) with the `auto` keyword, or you can omit it; the variable is still automatic and its value is destroyed when its local block ends.

Local static variables do not lose their values when their function ends. They remain local to that function. When the function is called after the first time, the static variable's value is still in place. You declare a static variable by placing the `static` keyword before the variable's definition.

Examples

1. Consider this program:

```
// Filename: C17STA1.CPP
// Tries to use a static variable
// without a static declaration.
#include <iostream.h>
triple_it(int ctr);

main()
{
   int ctr;                    // Used in the for loop to
                               // call a function 25 times.
   for (ctr=1; ctr<=25; ctr++)
     { triple_it(ctr); }        // Pass ctr to a function
                                    // called triple_it().
   return 0;
}

triple_it(int ctr)
{
   int total=0, ans;           // Local automatic variables.
```

```
    // Triples whatever value is passed to it
    // and adds the total.

    ans = ctr * 3;                  // Triple number passed.
    total += ans;  // Add triple numbers as this is called.

    cout << "The number " << ctr << " multiplied by 3 is "
         << ans << "\n";

    if (total > 300)
      { cout << "The total of triple numbers is over 300 \n"; }
    return 0;
}
```

This is a nonsense program that doesn't do much, yet you might sense something is wrong. The program passes numbers from 1 to 25 to the function called `triple_it`. The function triples the number and prints it.

The variable called `total` is initially set to 0. The idea here is to add each tripled number and print a message when the total is larger than 300. However, the `cout` never executes. For each of the 25 times that this subroutine is called, `total` is reset to 0. The `total` variable is an automatic variable, with its value erased and initialized every time its procedure is called. The next example corrects this.

2. If you want `total` to retain its value after the procedure ends, you must make it static. Because local variables are automatic by default, you have to include the `static` keyword to override this default. Then the value of the `total` variable is retained each time the subroutine is called.

   The following corrects the mistake in the previous program.

```
// Filename: C17STA2.CPP
// Uses a static variable with the static declaration.
#include <iostream.h>
triple_it(int ctr);

main()
```

```
{
   int ctr;                    // Used in the for loop to
                              //  call a function 25 times.
   for (ctr=1; ctr<=25; ctr++)
     { triple_it(ctr); }        // Pass ctr to a function
                                    // called triple_it().

   return 0;
}

triple_it(int ctr)
{
   static int total=0;               // Local and static
   int ans;                         // Local and automatic
   // total is set to 0 only the first time this
   // function is called.

   // Triples whatever value is passed to it and adds
   // the total.

   ans = ctr * 3;                  // Triple number passed.
   total += ans;  // Add triple numbers as this is called.

   cout << "The number " << ctr << " multiplied by 3 is "
          << ans << "\n";

   if (total > 300)
     { cout << "The total of triple numbers is over 300 \n"; }
   return 0;
}
```

This program's output follows. Notice that the function's `cout` is triggered, even though `total` is a local variable. Because `total` is static, its value is not erased when the function finishes. When `main()` calls the function a second time, `total`'s previous value (at the time you left the routine) is still there.

```
The number 1 multiplied by 3 is 3
The number 2 multiplied by 3 is 6
The number 3 multiplied by 3 is 9
The number 4 multiplied by 3 is 12
```

```
The number 5 multiplied by 3 is 15
The number 6 multiplied by 3 is 18
The number 7 multiplied by 3 is 21
The number 8 multiplied by 3 is 24
The number 9 multiplied by 3 is 27
The number 10 multiplied by 3 is 30
The number 11 multiplied by 3 is 33
The number 12 multiplied by 3 is 36
The number 13 multiplied by 3 is 39
The number 14 multiplied by 3 is 42
The number 15 multiplied by 3 is 45
The number 16 multiplied by 3 is 48
The number 17 multiplied by 3 is 51
The number 18 multiplied by 3 is 54
The number 19 multiplied by 3 is 57
The number 20 multiplied by 3 is 60
The number 21 multiplied by 3 is 63
The number 22 multiplied by 3 is 66
The number 23 multiplied by 3 is 69
The number 24 multiplied by 3 is 72
The number 25 multiplied by 3 is 75
```

This does not mean that local static variables become global. The main program cannot refer, use, print, or change `total` because it is local to the second function. Static simply means that the local variable's value is still there if the program calls the function again.

Three Issues of Parameter Passing

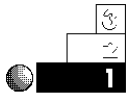
To have a complete understanding of programs with several functions, you have to learn three additional concepts:

- Passing arguments (variables) by value (also called "by copy")
- Passing arguments (variables) by address (also called "by reference")
- Returning values from functions

The first two concepts deal with the way local variables are passed and received. The third concept describes how receiving functions send values back to the calling functions. Chapter 18, "Passing Values," concludes this discussion by explaining these three methods for passing parameters and returning values.

Review Questions

The answers to the review questions are in Appendix B.



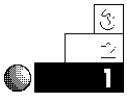
1. True or false: A function should always include a `return` statement as its last command, even though `return` is not required.
2. When a local variable is passed, is it called an argument or a parameter?

3. True or false: A function that is passed variables from another function cannot also have its own local variables.
4. What must appear inside the receiving function's parentheses, other than the variables passed to it?
5. If a function keeps track of a total or count every time it is called, should the counting or totaling variable be automatic or static?

6. When would you pass a global variable to a function? (Be careful—this might be a trick question!)
7. How many arguments are there in the following statement?

```
printf("The rain has fallen %d inches.", rainf);
```

Review Exercises



1. Write a program that asks, in `main()`, for the age of the user's dog. Write a second function called `people()` that computes the dog's age in human years (by multiplying the dog's age by seven).

2. Write a function that counts the number of times it is called. Name the function `count_it()`. Do not pass it anything. In the body of `count_it()`, print the following message:

   ```
   The number of times this function has been called is: ##
   ```

   where `##` is the number. (*Hint:* Because the variable must be local, make it static and initialize it to zero when you first define it.)

3. The following program contains several problems. Some of these problems produce errors. One problem is not an error, but a bad location for a variable declaration. (*Hint:* Find all the global variables.) See if you can spot some of the problems, and rewrite the program so it works better.

```
// Filename: C17BAD.CPP
// Program with bad uses of variable declarations.
#include <iostream.h>
#define NUM 10
do_var_fun();  // Prototypes discussed later.

char city[] = "Miami";
int count;

main()
{
   int abc;

   count = NUM;
   abc = 5;
   do_var_fun();

   cout << abc << " " << count << " " << pgm_var << " "
        << xyz;
   return 0;
}

int pgm_var = 7;

do_var_fun()
```

```
{
   char xyz = 'A';

   xyz = 'b';
   cout << xyz << " " << pgm_var << " " abc << " " << city;
   return 0;
}
```

Summary

Parameter passing is necessary because local variables are better than global. Local variables are protected in their own routines, but sometimes they must be shared with other routines. If local data are to remain in those variables (in case the function is called again in the same program), the variables should be static because otherwise their automatic values disappear.

Most the information in this chapter becomes more obvious as you use functions in your own programs. Chapter 18, "Passing Values," covers the actual passing of parameters in more detail and shows you two different ways to pass them.

Passing Values

C++ passes variables between functions using two different methods. The one you use depends on how you want the passed variables to be changed. This chapter explores these two methods. The concepts discussed here are not new to the C++ language. Other programming languages, such as Pascal, FORTRAN, and QBasic, pass parameters using similar techniques. A computer language must have the capability to pass information between functions before it can be called truly structured.

This chapter introduces you to the following:

- Passing variables by value
- Passing arrays by address
- Passing nonarrays by address

Pay close attention because most of the programs in the remainder of the book rely on the methods described in this chapter.

Passing by Value (by Copy)

The two wordings "passing by value" and "passing by copy" mean the same thing in computer terms. Some textbooks and C++ programmers state that arguments are passed *by value,* and some state that they are passed *by copy*. Both of these phrases describe one

of the two methods by which arguments are passed to receiving functions. (The other method is called "by address," or "by reference." This method is covered later in the chapter.)

When you pass by value, a copy of the variable's value is passed to the receiving function.

When an argument (local variable) is passed by value, a copy of the variable's value is sent to—and is assigned to—the receiving function's parameter. If more than one variable is passed by value, a copy of each of their values is sent to—and is assigned to—the receiving function's parameters.

Figure 18.1 shows the *passing by copy* in action. The value of `i`—not the variable—is passed to the called function, which receives it as a variable `i`. There are two variables called `i`, not one. The first is local to `main()`, and the second is local to `pr_it()`. They both have the same names, but because they are local to their respective functions, there is no conflict. The variable does not have to be called `i` in both functions, and because the value of `i` is sent to the receiving function, it does not matter what the receiving function calls the variable that receives this value.

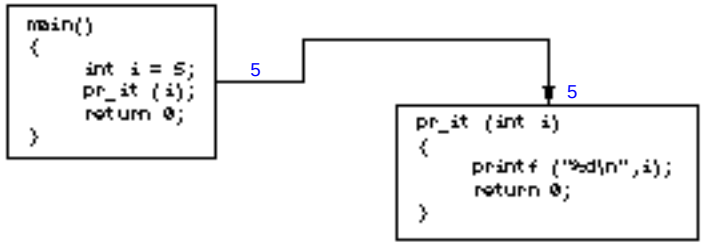


Figure 18.1. Passing the variable `i` by value.

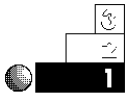
In this case, when passing and receiving variables between functions, it is wisest to retain the same names. Even though they are not the same variables, they hold the same value. In this example, the value `5` is passed from `main()`'s `i` to `pr_it()`'s `i`.

Because a copy of `i`'s value (and not the variable itself) is passed to the receiving function, if `pr_it()` changed `i`, it would be changing only its copy of `i` and not `main()`'s `i`. This fact truly separates functions and variables. You now have the technique for passing a copy of a variable to a receiving function, with the receiving function being unable to modify the calling function's variable.

All C++'s nonarray variables you have seen so far are passed by value. You do not have to do anything special to pass variables by value, except to pass them in the calling function's argument list and receive them in the receiving function's parameter list.

NOTE: The default method for passing parameters is by value, as just described, unless you pass arrays. Arrays are always passed by the other method, by address, described later in the chapter.

Examples



1. The following program asks users for their weight. It then passes that weight to a function that calculates the equivalent weight on the moon. Notice the second function uses the passed value, and calculates with it. After `weight` is passed to the second function, that function can treat `weight` as though it were a local variable.

Identify the program and include the necessary input/output file.

You want to calculate the user's weight on the moon. Because you have to hold the user's weight somewhere, declare the variable `weight` *as an integer. You also need a function that does the calculations, so create a function called* `moon()`.

Ask the user how much he or she weighs. Put the user's answer in `weight`. *Now pass the user's weight to the* `moon()` *function, which divides the weight by six to determine the equivalent weight on the moon. Display the user's weight on the moon.*

You have finished, so leave the `moon()` *function, then leave the* `main()` *function.*

```
// Filename: C18PASS1.CPP
// Calculate the user's weight in a second function.
#include <iostream.h>
moon(int weight);  // Prototypes discussed later.
```

```
main()
{
   int weight;                    // main()'s local weight.
   cout << "How many pounds do you weigh? ";
   cin >> weight;

   moon(weight);          // Call the moon() function and
                          //   pass it the weight.
   return 0;              // Return to the operating system.
}

moon(int weight)           // Declare the passed parameter.
{
   // Moon weights are 1/6th earth's weights
   weight /= 6;                  // Divide the weight by six.

   cout << "You weigh only " << weight <<
           " pounds on the moon!";
   return 0;                               // Return to main().
}
```

The output of this program follows:

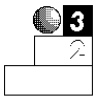
```
How many pounds do you weigh? 120
You weigh only 20 pounds on the moon!
```

2. You can rename passed variables in the receiving function. They are distinct from the passing function's variable. The following is the same program as in Example 1, except the receiving function calls the passed variable `earth_weight`. A new variable, called `moon_weight`, is local to the called function and is used for the moon's equivalent weight.


```
// Filename: C18PASS2.CPP
// Calculate the user's weight in a second function.
#include <iostream.h>
moon(int earth_weight);

main()
```

```
{
   int weight;                    // main()'s local weight.
   cout << "How many pounds do you weigh? ";
   cin >> weight;

   moon(weight);          // Call the moon() function and
                          // pass it the weight.
   return 0;              // Return to the operating system.
}

moon(int earth_weight)    // Declare the passed parameter.
{
   int moon_weight;               // Local to this function.

   // Moon's weights are 1/6th of earth's weights.
   moon_weight = earth_weight / 6;  // Divide weight by six.

   cout << "You only weigh " << moon_weight <<
           " pounds on the moon!";
   return 0;                                // Return to main().
}
```

The resulting output is identical to that of the previous program. Renaming the passed variable changes nothing.



3. The next example passes three variables—of three different types—to the called function. In the receiving function's parameter list, each of these variable types must be declared.

   This program prompts users for three values in the `main()` function. The `main()` function then passes these variables to the receiving function, which calculates and prints values related to those passed variables. When the called function modifies a variable passed to the function, notice again that this does not affect the calling function's variable. When variables are passed by value, the value—not the variable—is passed.

```
// Filename: C18PASS3.CPP
// Get grade information for a student.
#include <iostream.h>
#include <iomanip.h>
check_grade(char lgrade, float average, int tests);
```

```
main()
{
   char lgrade;   // Letter grade.
   int  tests;    // Number of tests not yet taken.
   float average; // Student's average based on 4.0 scale.

   cout << "What letter grade do you want? ";
   cin >> lgrade;
   cout << "What is your current test average? ";
   cin >> average;
   cout << "How many tests do you have left? ";
   cin >> tests;

   check_grade(lgrade, average, tests);  // Calls function
                  //  and passes three variables by value.
   return 0;
}

check_grade(char lgrade, float average, int tests)
{
   switch (tests)
   {
     case (0): { cout << "You will get your current grade "
                      << "of " << lgrade;
                 break; }
     case (1): { cout << "You still have time to bring " <<
                         "up your average";
                 cout << "of " << setprecision(1) <<
                         average << "up. Study hard!";
                 break; }
     default:  { cout << "Relax. You still have plenty of "
                      <<  "time.";
                 break; }
   }
   return 0;
}
```

Passing by Address (by Reference)

When you pass by address, the address of the variable is passed to the receiving function.

The two phrases "by address" and "by reference" mean the same thing. The previous section described passing arguments by value (or by copy). This section teaches you how to pass arguments by address.

When you pass an argument (local variable) *by address*, the variable's address is sent to—and is assigned to—the receiving function's parameter. (If you pass more than one variable by address, each of their addresses is sent to—and is assigned to—the receiving function's parameters.)

Variable Addresses

All variables in memory (RAM) are stored at memory addresses—see Figure 18.2. If you want more information on the internal representation of memory, refer to Appendix A, "Memory Addressing, Binary, and Hexadecimal Review."

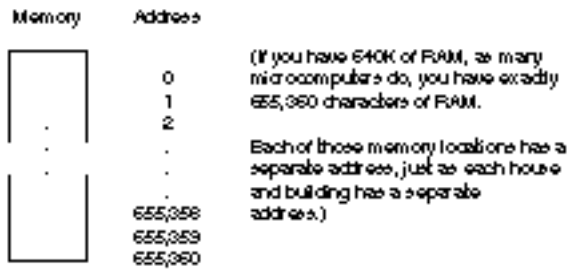


Figure 18.2. Memory addresses.

When you tell C++ to define a variable (such as `int i;`), you are requesting C++ to find an unused place in memory and assign that place (or memory address) to `i`. When your program uses the variable called `i`, C++ goes to `i`'s address and uses whatever is there.

If you define five variables as follows,

```
int i;
float x=9.8;
char ara[2] = {'A', 'B'};
int j=8, k=3;
```

C++ might arbitrarily place them in memory at the addresses shown in Figure 18.3.

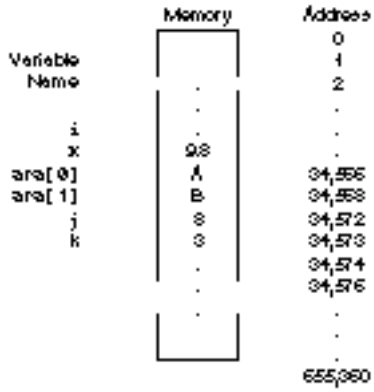


Figure 18.3. Storing variables in memory.

You don't know what is contained in the variable called `i` because you haven't put anything in it yet. Before you use `i`, you should initialize it with a value. (All variables—except character variables—usually use more than 1 byte of memory.)

Sample Program

All C++ arrays are passed by address.

The address of the variable, not its value, is copied to the receiving function when you pass a variable by address. In C++, *all arrays are automatically passed by address.* (Actually, a copy of their address is passed, but you will understand this better when you learn more about arrays and pointers.) The following important rule holds true for programs that pass by address:

Every time you pass a variable by address, if the receiving function changes the variable, it is changed also in the calling function.

Therefore, if you pass an array to a function and the function changes the array, those changes are still with the array when it returns to the calling function. Unlike passing by value, passing by address gives you the ability to change a variable in the *called* function and to keep those changes in effect in the *calling* function. The following sample program helps to illustrate this concept.

```
// Filename: C18ADD1.CPP
// Passing by address example.
#include <iostream.h>
#include <string.h>
change_it(char c[4]);   // Prototype discussed later.
main()
{
   char name[4]="ABC";

   change_it(name);          // Passes by address because
                             // it is an array.
   cout << name << "\n";     // Called function can
                             // change array.
   return 0;
}

change_it(char c[4])         // You must tell the function
                             // that c is an array.
{
   cout << c << "\n";        // Print as it is passed.
   strcpy(c, "USA");         // Change the array, both
                             // here and in main().
   return 0;
}
```

Here is the output from this program:

```
ABC
USA
```

At this point, you should have no trouble understanding that the array is passed from `main()` to the function called `change_it()`. Even though `change_it()` calls the array `c`, it refers to the same array passed by the `main()` function (`name`).

Figure 18.4 shows how the array is passed. Although the address of the array—and not its value—is passed from `name` to `c`, both arrays are the same.

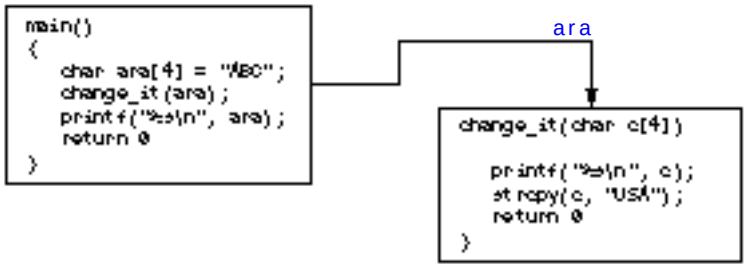


Figure 18.4. Passing an array by address.

Before going any further, a few additional comments are in order. Because the address of `name` is passed to the function—even though the array is called `c` in the receiving function—it is still the same array as `name`. Figure 18.5 shows how C++ accomplishes this task at the memory-address level.

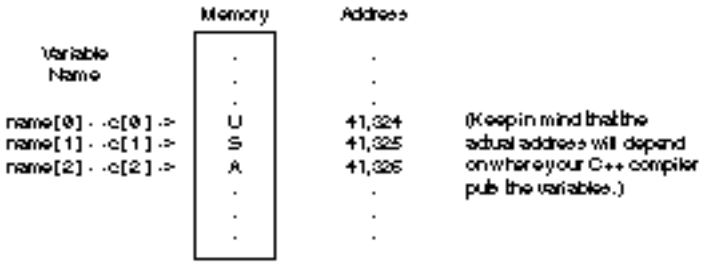
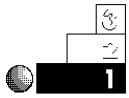


Figure 18.5. The array being passed is the same array in both functions.

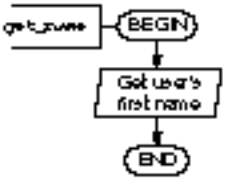
The variable array is referred to as `name` in `main()` and as `c` in `change_it()`. Because the address of `name` is copied to the receiving function, the variable is changed no matter what it is called in either

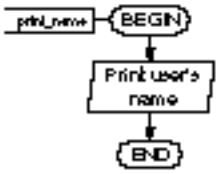
function. Because `change_it()` changes the array, the array is changed also in `main()`.

Examples



1. You can now use a function to fill an array with user input. The following function asks users for their first name in the function called `get_name()`. As users type the name in the array, it is also entered in `main()`'s array. The `main()` function then passes the array to `pr_name()`, where it is printed. (If arrays were passed by value, this program would not work. Only the array value would be passed to the called functions.)






```
// Filename: C18ADD2.CPP
// Get a name in an array, then print it using
// separate functions.
#include <iostream.h>
get_name(char name[25]);   // Prototypes discussed later.
print_name(char name[25]);

main()
{
   char name[25];
   get_name(name);           // Get the user's name.
   print_name(name);         // Print the user's name.
   return 0;
}

get_name(char name[25])      // Pass the array by address.
{
   cout << "What is your first name? ";
   cin >> name;
   return 0;
}

print_name(char name[25])
{
   cout << "\n\n Here you are, " << name;
   return 0;
}
```

When you pass an array, be sure to specify the array's type in the receiving function's parameter list. If the previous program declared the passed array with

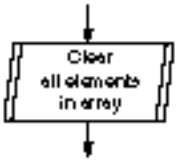
```
get_name(char name)
```

the function `get_name()` would interpret this as a single character variable, *not* a character array. You never have to put the array size in brackets. The following statement also works as the first line of `get_name()`.

```
get_name(char name[])
```

Most C++ programmers put the array size in the brackets to clarify the array size, even though the size is not needed.

2. Many programmers pass character arrays to functions to erase them. Here is a function called `clear_it()`. It expects two parameters: a character array and the total number of elements declared for that array. The array is passed by address (as are all arrays) and the number of elements, `num_els`, is passed by value (as are all nonarrays). When the function finishes, the array is cleared (all its elements are reset to null zero). Subsequent functions that use it can then have an empty array.




```
clear_it(char ara[10], int num_els)
{
   int ctr;
   for (ctr=0; ctr<num_els; ctr++)
     { ara[ctr] = '\0'; }
   return 0;
}
```

The brackets after `ara` do not have to contain a number, as described in the previous example. The `10` in this example is simply a placeholder for the brackets. Any value (or no value) would work as well.

Passing Nonarrays by Address

You can pass nonarrays by address as well.

You now should see the difference between passing variables by address and by value. Arrays can be passed by address, and nonarrays can be passed by value. You can override the *by value* default for nonarrays. This is helpful sometimes, but it is not always recommended because the called function can damage values in the called function.

If you want a nonarray variable changed in a receiving function and also want the changes kept in the calling function, you must override the default and pass the variable by address. (You should understand this section better after you learn how arrays and pointers relate.) To pass a nonarray by address, you must precede the argument in the receiving function with an ampersand (`&`).

This might sound strange to you (and it is, at this point). Few C++ programmers override the default of passing by address. When you learn about pointers later, you should have little need to do so. Most C++ programmers don't like to clutter their code with these extra ampersands, but it's nice to know you can override the default if necessary.

The following examples demonstrate how to pass nonarray variables by address.

Examples

1. The following program passes a variable by address from `main()` to a function. The function changes it and returns to `main()`. Because the variable is passed by address, `main()` recognizes the new value.

```
// Filename: C18ADD3.CPP
// Demonstrate passing nonarrays by address.
#include <iostream.h>
do_fun(int &amt);   // Prototypes discussed later.

main()
{
   int amt;
```

```
   amt = 100;                  // Assign a value in main().
   cout << "In main(), amt is " << amt << "\n";

   do_fun(amt);          // Pass amt by address
   cout << "After return, amt is " << amt << " in main()\n";
   return 0;
}

do_fun(int &amt)               // Inform function of
                               // passing by address.
{
   amt = 85;                   // Assign new value to amt.
   cout << "In do_fun(), amt is " << amt << "\n";
   return 0;
}
```

The output from this program follows:

```
In main(), amt is 100
In do_fun(), amt is 85
After return, amt is 85 in main()
```

Notice that `amt` changed in the called function. Because it was passed by address, it is changed also in the calling function.

2. You can use a function to get the user's keyboard values. The `main()` function recognizes those values as long as you pass them by address. The following program calculates the cubic feet in a swimming pool. In one function, it requests the width, length, and depth. In another function, it calculates the cubic feet of water. Finally, in a third function, it prints the answer. The `main()` function is clearly a controlling function, passing variables between these functions by address.


```
// Filename: C18POOL.CPP
// Calculates the cubic feet in a swimming pool.
#include <iostream.h>
get_values(int &length, int &width, int &depth);
calc_cubic(int &length, int &width, int &depth, int &cubic);
print_cubic(int &cubic);
```

```
main()
{
   int length, width, depth, cubic;

   get_values(length, width, depth);
   calc_cubic(length, width, depth, cubic);
   print_cubic(cubic);

   return 0;
}

get_values(int &length, int &width, int &depth)
{
   cout << "What is the pool's length? ";
   cin >> length;
   cout << "What is the pool's width? ";
   cin >> width;
   cout << "What is the pool's average depth? ";
   cin >> depth;
   return 0;
}

calc_cubic(int &length, int &width, int &depth, int &cubic)
{
   cubic = (length) * (width) * (depth);
   return 0;
}

print_cubic(int &cubic)
{
   cout << "\nThe pool has " << cubic << " cubic feet\n";
   return 0;
}
```

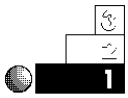
The output follows:

```
What is the pool's length? 16
What is the pool's width? 32
What is the pool's average depth? 6
The pool has 3072 cubic feet
```

All variables in a function must be preceded with an ampersand if they are to be passed by address.

Review Questions

The answers to the review questions are in Appendix B.



1. What type of variable is automatically passed by address?
2. What type of variable is automatically passed by value?
3. True or false: If a variable is passed by value, it is passed also by copy.

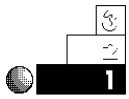
4. If a variable is passed to a function by value and the function changes the variable, is it changed in the calling function?
5. If a variable is passed to a function by address and the function changes the variable, is it changed in the calling function?

6. What is wrong with the following function?

```
do_fun(x, y, z)
{
   cout << "The variables are " << x << y << z;
   return 0;
}
```

7. Suppose you pass a nonarray variable and an array to a function at the same time. What is the default?
   a. Both are passed by address.
   b. Both are passed by value.
   c. One is passed by address and the other is passed by value.

Review Exercises



1. Write a `main()` function and a second function that `main()` calls. Ask users for their annual income in `main()`. Pass the income to the second function and print a congratulatory message if the user makes more than $50,000 or an encouragement message if the user makes less.

2. Write a three-function program, consisting of the following functions:

```
main()
fun1()
fun2()
```

   Declare a 10-element character array in `main()`, fill it with the letters `A` through `J` in `fun1()`, then print that array backwards in `fun2()`.

3. Write a program whose `main()` function passes a number to a function called `print_aster()`. The `print_aster()` function prints that many asterisks on a line, across the screen. If `print_aster()` is passed a number greater than 80, display an error because most screens cannot print more than 80 characters on the same line. When execution is finished, return control to `main()` and then return to the operating system.

4. Write a function that is passed two integer values by address. The function should declare a third local variable. Use the third variable as an intermediate variable and swap the values of both passed integers. For example, suppose the calling function passes your function `old_pay` and `new_pay` as in

```
swap_it(old_pay, new_pay);
```

   The `swap_it()` function reverses the two values so, when control returns to the calling function, the values of `old_pay` and `new_pay` are swapped.

Summary

You now have a complete understanding of the various methods for passing data to functions. Because you will be using local variables as much as possible, you have to know how to pass local variables between functions but also keep the variables away from functions that don't need them.

You can pass data in two ways: by value and by address. When you pass data by value, which is the default method for nonarrays, only a copy of the variable's contents are passed. If the called function modifies its parameters, those variables are not modified in the calling function. When you pass data by address, as is done with arrays and nonarray variables preceded by an ampersand, the receiving function can change the data in both functions.

Whenever you pass values, you must ensure that they match in number and type. If you don't match them, you could have problems. For example, suppose you pass an array and a floating-point variable, but in the receiving function, you receive a floating-point variable followed by an array. The data does not reach the receiving function properly because the parameter data types do not match the variables being passed. Chapter 19, "Function Return Values and Prototypes," shows you how to protect against such disasters by prototyping all your functions.

Function Return Values and Prototypes

So far, you have passed variables to functions in only one direction—a calling function passed data to a receiving function. You have yet to see how data are passed back *from* the receiving function to the calling function. When you pass variables by address, the data are changed in both functions—but this is different from passing data back. This chapter focuses on writing function return values that improve your programming power.

After you learn to pass and return values, you have to *prototype* your own functions as well as C++'s built-in functions, such as `cout` and `cin`. By prototyping your functions, you ensure the accuracy of passed and returned values.

This chapter introduces you to the following:

- Returning values from functions
- Prototyping functions
- Understanding header files

By returning values from functions, you make your functions fully modular. They can now stand apart from the other functions.

They can receive and return values and act as building blocks that compose your complete application.

Function Return Values

Until now, all functions in this book have been *subroutines* or *subfunctions*. A C++ subroutine is a function that is called from another function, but it does not return any values. The difference between subroutines and functions is not as critical in C++ as it is in other languages. All functions, whether they are subroutines or functions that return values, are defined in the same way. You can pass variables to each of them, as you have seen throughout this section of the book.

Put the return value at the end of the `return` statement.

Functions that return values offer you a new approach to programming. In addition to passing data one-way, from calling to receiving function, you can pass data back from a receiving function to its calling function. When you want to return a value from a function to its calling function, put the return value after the `return` statement. To clarify the return value even more, many programmers put parentheses around the return value, as shown in the following syntax:

```
return (return value);
```

CAUTION: Do not return global variables. There is no need to do so because their values are already known throughout the code.

The calling function must have a use for the return value. For example, suppose you wrote a function that calculated the average of any three integer variables passed to it. If you return the average, the calling function has to receive that return value. The following sample program helps to illustrate this principle.


```
// Filename: C19AVG.CPP
// Calculates the average of three input values.
#include <iostream.h>
int calc_av(int num1, int num2, int num3);   //Prototype
```

```
main()
{
   int num1, num2, num3;
   int avg;                  // Holds the return value.

   cout << "Please type three numbers (such as 23 54 85) ";
   cin >> num1 >> num2 >> num3;

   // Call the function, pass the numbers,
   // and accept the return value amount.
   avg = calc_av(num1, num2, num3);

   cout << "\n\nThe average is " << avg;      // Print the
                                              // return value.
   return 0;
}

int calc_av(int num1, int num2, int num3)
{
   int local_avg;  // Holds the average for these numbers.
   local_avg = (num1+num2+num3) / 3;

   return (local_avg);
}
```

Here is a sample output from the program:

```
Please type three numbers (such as 23 54 85) 30 40 50


The average is 40
```

Study this program carefully. It is similar to many you have seen, but a few additional points have to be considered now that the function returns a value. It might help to walk through this program a few lines at a time.

The first part of `main()` is similar to other programs you have seen. It declares its local variables: three for user input and one for the calculated average. The `cout` and `cin` are familiar to you. The function call to `calc_av()` is also familiar; it passes three variables

(num1, num2, and num3) by value to calc_av(). (If it passed them by address, an ampersand (&) would have to precede each argument, as discussed in Chapter 18.)

Put the function's return type before its name. If you don't specify a return type, int is the default.

The receiving function, calc_av(), seems similar to others you have seen. The only difference is that the first line, the function's definition line, has one addition—the int before its name. This is the *type* of the return value. You must always precede a function name with its return data type. If you do not specify a type, C++ assumes a type of int. Therefore, if this example had no return type, it would work just as well because an int return type would be assumed.

Because the variable being returned from calc_av() is an integer, the int return type is placed before calc_av()'s name.

You can see also that the return statement of calc_av() includes the return value, local_avg. This is the variable being sent back to the calling function, main(). You can return only a single variable to a calling function.

Even though a function can receive more than one parameter, it can return only a single value to the calling function. If a receiving function is modifying more than one value from the calling function, you must pass the parameters by address; you cannot return multiple values using a return statement.

After the receiving function, calc_av(), returns the value, main() must do something with that returned value. So far, you have seen function calls on lines by themselves. Notice in main() that the function call appears on the right side of the following assignment statement:

```
avg = calc_av(num1, num2, num3);
```

When the calc_av() function returns its value—the average of the three numbers—that value replaces the function call. If the average computed in calc_av() is 40, the C++ compiler interprets the following statement in place of the function call:

```
avg = 40;
```

You typed a function call to the right of the equal sign, but the program replaces a function call with its return value when the return takes place. In other words, a function that returns a value

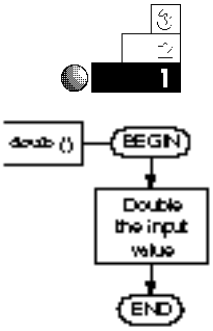
becomes that value. You must put such a function anywhere you put any variable or literal (usually to the right of an equal sign, in an expression, or in `cout`). The following is an *incorrect* way of calling `calc_av()`:

```
calc_av(num1, num2, num3);
```

If you did this, C++ would have nowhere to put the return value.

CAUTION: Function calls that return values usually don't appear on lines by themselves. Because the function call is replaced by the return value, you should do something with that return value (such as assign it to a variable or use it in an expression). Return values can be ignored, but doing so usually defeats the purpose of creating them.

Examples



1. The following program passes a number to a function called `doub()`. The function doubles the number and returns the result.

```
// Filename: C19DOUB.CPP
// Doubles the user's number.
#include <iostream.h>
int doub (int num);
main()
{
   int number;                        // Holds user's input.
   int d_number;     // Holds double the user's input.
   cout << "What number do you want doubled? ";
   cin >> number;

   d_number = doub(number);         // Assigns return value.
   cout << number << " doubled is " << d_number;
   return 0;
}
```

```
int doub(int num)
{
   int d_num;
   d_num = num * 2;                    // Doubles the number.
   return (d_num);                     // Returns the result.
}
```

The program produces output such as this:

```
What number do you want doubled? 5
5 doubled is 10
```

2. Function return values can be used *anywhere* literals, variables, and expressions are used. The following program is similar to the previous one. The difference is in `main()`.

   The function call is performed not on a line by itself, but from a `cout`. This is a nested function call. You call the built-in function `cout` using the return value from one of the program's functions named `doub()`. Because the call to `doub()` is replaced by its return value, the `cout` has enough information to proceed as soon as `doub()` returns. This gives `main()` less overhead because it no longer needs a variable called `d_number`, although you must use your own judgment as to whether this program is easier to maintain. Sometimes it is wise to include function calls in other expressions; other times it is clearer to call the function and assign its return value to a variable before using it.

```
// Filename: C19DOUB2.CPP
// Doubles the user's number.
#include <iostream.h>
int doub(int num); // Prototype

main()
{
   int number;                         // Holds user's input.
   cout << "What number do you want doubled? ";
   cin >> number;
```

```
   // The third cout parameter is
   // replaced with a return value.
   cout << number << " doubled is " << doub(number);

   return 0;
}

int doub(int num)
{
   int d_num;
   d_num = num * 2;                    // Double the number.
   return (d_num);                     // Return the result.
}
```

3. The following program asks the user for a number. That number is then passed to a function called `sum()`, which adds the numbers from `1` to that number. In other words, if the user types a `6`, the function returns the result of the following calculation:

```
1 + 2 + 3 + 4 + 5 + 6
```

This is known as the *sum of the digits* calculation, and it is sometimes used for depreciation in accounting.

```
// Filename: C19SUMD.CPP
// Compute the sum of the digits.
#include <iostream.h>
int sum(int num);  // Prototype

main()
{
   int num, sumd;

   cout << "Please type a number: ";
   cin >> num;

   sumd = sum(num);
   cout << "The sum of the digits is " << sumd;
   return 0;
}
```

```
int sum(int num)
{
   int ctr;                          // Local loop counter.
   int sumd=0;                    // Local to this function.
   if (num <= 0)  // Check whether parameter is too small.
     { sumd = num; }    // Returns parameter if too small.
   else
     { for (ctr=1; ctr<=num; ctr++)
          { sumd += ctr; }
     }
   return(sumd);
}
```

The following is a sample output from this program:

```
Please type a number: 6
The sum of the digits is 21
```

4. The following program contains two functions that return values. The first function, `maximum()`, returns the larger of two numbers entered by the user. The second one, `minimum()`, returns the smaller.

```
// Filename: C19MINMX.CPP
// Finds minimum and maximum values in functions.
#include <iostream.h>

int maximum(int num1, int num2);  // Prototypes
int minimum(int num1, int num2);

main()
{
   int num1, num2;                  // User's two numbers.
   int min, max;

   cout << "Please type two numbers (such as 46 75) ";
   cin >> num1 >> num2;

   max = maximum(num1, num2);      // Assign the return
   min = minimum(num1, num2);      // value of each
                                   // function to variables.
```

```
    cout << "The minimum number is " << min << "\n";
    cout << "The maximum number is " << max << "\n";
    return 0;
}

int maximum(int num1, int num2)
{
    int max;                // Local to this function only.
    max = (num1 > num2) ? (num1) : (num2);
    return (max);
}

int minimum(int num1, int num2)
{
    int min;                // Local to this function only.
    min = (num1 < num2) ? (num1) : (num2);
    return (min);
}
```

Here is a sample output from this program:

```
Please type two numbers (such as 46 75) 72 55
The minimum number is 55
The maximum number is 72
```

If the user types the same number, `minimum` and `maximum` are the same.

These two functions can be passed any two integer values. In such a simple example as this one, the user certainly already knows which number is lower or higher. The point of such an example is to show how to code return values. You might want to use similar functions in a more useful application, such as finding the highest paid employee from a payroll disk file.

Function Prototypes

The word *prototype* is sometimes defined as a model. In C++, a function prototype models the actual function. Before completing

your study of functions, parameters, and return values, you must understand how to prototype each function in your program.

C++ requires that you prototype all functions in your program. When prototyping, you inform C++ of the function's parameter types and its return value, if any.

To prototype a function, copy the function's definition line to the top of your program (immediately before or after the `#include <iostream.h>` line). Place a semicolon at the end of the function definition line, and you have the prototype. The definition line (the function's first line) contains the return type, the function name, and the type of each argument, so the function prototype serves as a model of the function that follows.

If a function does not return a value, or if that function has no arguments passed to it, you should still prototype it. Place the keyword `void` in place of the return type or the parameters. `main()` is the only function that you do not have to prototype because it is *self-prototyping;* meaning `main()` is not called by another function. The first time `main()` appears in your program (assuming you follow the standard approach and make `main()` your program's first function), it is executed.

If a function returns nothing, `void` must be its return type. Put `void` in the argument parentheses of function prototypes with no arguments. All functions must match their prototypes.

C++ assumes functions return `int` unless you put a different data return type, or use the `void` keyword.

All `main()` functions in this book have returned a `0`. Why? You now know enough to answer that question. Because `main()` is self-prototyping, and because the `void` keyword never appeared before `main()` in these programs, C++ assumed an `int` return type. All C++ functions prototyped as returning `int` or those without any return data type prototype assume `int`. If you wanted to not put `return 0;` at the end of `main()`'s functions, you must insert `void` before `main()` as in:

```
void main()     // main() self-prototypes to return nothing.
```

You can look at a statement and tell whether it is a prototype or a function definition (the function's first line) by the semicolon on the end. All prototypes, unless you make `main()` self-prototype, end with a semicolon.

Prototype for Safety

Prototyping protects you from programming mistakes. Suppose you write a function that expects two arguments: an integer followed by a floating-point value. Here is the first line of such a function:

```
my_fun(int num, float amount)
```

What if you passed incorrect data types to `my_fun()`? If you were to call this function by passing it two literals, a floating-point followed by an integer, as in

```
my_fun(23.43, 5);          // Call the my_fun() function.
```

the function would not receive correct parameters. It is expecting an integer followed by a floating-point, but you did the opposite and sent it a floating-point followed by an integer.

Prototyping protects your programs from function programming errors.

In regular C programs, mismatched arguments such as these generate no error message even though the data are not passed correctly. C++ requires prototypes so you cannot send the wrong data types to a function (or expect the wrong data type to be returned). Prototyping the previous function results in this:

```
void my_fun(int num, float amount);        // Prototype
```

In doing so, you tell the compiler to check this function for accuracy. You inform the compiler to expect nothing after the `return` statement, not even `0`, (due to the `void` keyword) and to expect an integer followed by a floating-point in the parentheses.

If you break any of the prototype's rules, the compiler informs you of the problem and you can correct it.

Prototype All Functions

You should prototype every function in your program. As just described, the prototype defines (for the rest of the program) which functions follow, their return types, and their parameter types. You should prototype C++'s built-in functions also, such as `printf()` and `scanf()` if you use them.

Think about how you prototype printf(). You don't always pass it the same types of parameters because you print different data with each printf(). Prototyping functions you write is easy: The prototype is basically the first line in the function. Prototyping functions you do not write might seem difficult, but it isn't—you have already done it with every program in this book!

Header files contain built-in function prototypes.

The designers of C++ realized that all functions have to be prototyped. They realized also that you cannot prototype built-in functions, so they did it for you and placed the prototypes in header files on your disk. You have been including the printf() and scanf() prototypes in each program that used them in this book with the following statement:

```
#include <stdio.h>
```

Inside the stdio.h file is a prototype of many of C++'s input and output functions. By having prototypes of these functions, you ensure that they cannot be passed bad values. If someone attempts to pass incorrect values, C++ catches the problem.

Because printf() and scanf() are not used very often in C++, the cout and cin operators have their own header file called iostream.h that you have seen included in this book's programs as well. The iostream.h file does not actually include prototypes for cout and cin because they are operators and not functions, but iostream.h does include some needed definitions to make cout and cin work.

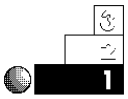
Remember too that iomanip.h has to be included if you use a setw or setprecision modifier in cout. Any time you use a new built-in C++ function or a manipulating operator, check your compiler's manual to find the name of the prototype file to include.

Prototyping is the primary reason why you should always include the matching header file when you use C++'s built-in functions. The strcpy() function you saw in previous chapters requires the following line:

```
#include <string.h>
```

This is the header file for the strcpy() function. Without it, the program does not work.

Examples

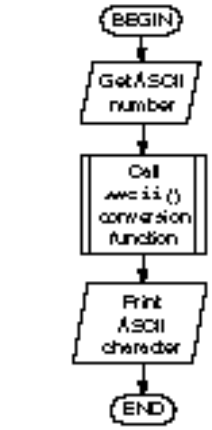


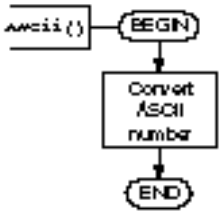
1. Prototype all functions in all programs except `main()`. Even `main()` must be prototyped if it returns nothing (not even `0`). The following program includes two prototypes: one for `main()` because it returns nothing, and one for the built-in `printf()` and `scanf()` functions.

```
// Filename: C19PRO1.CPP
// Calculates sales tax on a sale
#include <stdio.h>        // Prototype built-in functions.
void main(void);

void main(void)
{

   float total_sale;
   float tax_rate = .07;            // Assume seven percent
                                    // tax rate.

   printf("What is the sale amount? ");
   scanf(" %f", &total_sale);

   total_sale += (tax_rate * total_sale);
   printf("The total sale is %.2f", total_sale);
   return;      // No 0 required!
}
```

Notice that `main()`'s `return` statement needed only a semicolon after it. As long as you prototype `main()` with a `void` return type, the last line in `main()` can be `return;` instead of having to type `return 0;` each time.

2. The following program asks the user for a number in `main()`, and passes that number to `ascii()`. The `ascii()` function returns the ASCII character that matches the user's number. This example illustrates a `character` return type. Functions can return any data type.

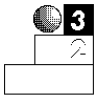





```
// Filename: C19ASC.CPP
// Prints the ASCII character of the user's number.
// Prototypes follow.
#include <iostream.h>
char ascii(int num);

void main()
{
   int num;
   char asc_char;

   cout << "Enter an ASCII number? ";
   cin >> num;

   asc_char = ascii(num);
   cout << "The ASCII character for " << num
        << " is " << asc_char;
   return;
}

char ascii(int num)
{
   char asc_char;
   asc_char = char(num);  // Type cast to a character.
   return (asc_char);
}
```

The output from this program follows:

```
Enter an ASCII number? 67
The ASCII character for 67 is C
```

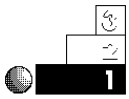


3. Suppose you have to calculate net pay for a company. You find yourself multiplying the hours worked by the hourly pay, then deducting taxes to compute the net pay. The following program includes a function that does this for you. It requires three arguments: the hours worked, the hourly pay, and the tax rate (as a floating-point decimal, such as .30 for 30 percent). The function returns the net pay. The `main()` calling program tests the function by sending three different payroll values to the function and printing the three return values.

```
// Filename: C19NPAY.CPP
// Defines a function that computes net pay.
#include <iostream.h>  // Needed for cout and cin.
void main(void);
float netpayfun(float hours, float rate, float taxrate);

void main(void)
{
   float net_pay;

   net_pay = netpayfun(40.0, 3.50, .20);
   cout << "The pay for 40 hours at $3.50/hr., and a 20% "
        << "tax rate is $";
   cout << net_pay << "\n";

   net_pay = netpayfun(50.0, 10.00, .30);
   cout << "The pay for 50 hours at $10.00/hr., and a 30% "
        << "tax rate is $";
   cout << net_pay << "\n";

   net_pay = netpayfun(10.0, 5.00, .10);
   cout << "The pay for 10 hours at $5.00/hr., and a 10% "
        << " tax rate is $";
   cout << net_pay << "\n";

   return;
}

float netpayfun(float hours, float rate, float taxrate)
{
   float gross_pay, taxes, net_pay;
   gross_pay = (hours * rate);
   taxes = (taxrate * gross_pay);
   net_pay = (gross_pay - taxes);
   return (net_pay);
}
```

Review Questions

The answers to the review questions are in Appendix B.



1. How do you declare function return types?
2. What is the maximum number of return values a function can return?
3. What are header files for?

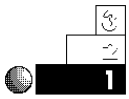
4. What is the default function return type?
5. True or false: a function that returns a value can be passed only a single parameter.

6. How do prototypes protect the programmer from bugs?
7. Why don't you have to return global variables?
8. What is the return type, given the following function prototype?

```
float my_fun(char a, int b, float c);
```

How many parameters are passed to `my_fun()`? What are their types?

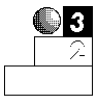
Review Exercises



1. Write a program that contains two functions. The first function returns the square of the integer passed to it, and the second function returns the cube. Prototype `main()` so you do not have to return a value.

2. Write a function that returns the double-precision area of a circle, given that a double-precision radius is passed to it. The formula for calculating the area of a circle is

```
area = 3.14159 * (radius * radius)
```



3. Write a function that returns the value of a polynomial given this formula:

```
9x4 + 15x2 + x1
```

Assume `x` is passed from `main()` and it is supplied by the user.

Summary

You learned how to build your own collection of functions. When you write a function, you might want to use it in more than one program—there is no need to reinvent the wheel. Many programmers write useful functions and use them in more than one program.

You now understand the importance of prototyping functions. You should prototype all your own functions, and include the appropriate header file when you use one of C++'s built-in functions. Furthermore, when a function returns a value other than an integer, you must prototype so C++ recognizes the noninteger return value.

Default Arguments and Function Overloading

All functions that receive arguments do not have to be sent values. C++ enables you to specify default argument lists. You can write functions that assume argument values even if you do not pass them any arguments.

C++ also enables you to write more than one function with the same function name. This is called *overloading functions.* As long as their argument lists differ, the functions are differentiated by C++.

This chapter introduces you to the following:

- Default argument lists
- Overloaded functions
- Name-mangling

Default argument lists and overloaded functions are not available in regular C. C++ extends the power of your programs by providing these time-saving procedures.

Default Argument Lists

Suppose you were writing a program that has to print a message on-screen for a short period of time. For instance, you pass a function an error message stored in a character array and the function prints the error message for a certain period of time.

The prototype for such a function can be this:

```
void pr_msg(char note[]);
```

Therefore, to request that `pr_msg()` print the line `"Turn printer on"`, you call it this way:

```
pr_msg("Turn printer on");    // Passes a message to be printed.
```

This command prints the message `"Turn printer on"` for a period of five seconds or so. To request that `pr_msg()` print the line `"Press any key to continue..."`, you call it this way:

```
pr_msg("Press a key to continue...");    // Passes a message.
```

As you write more of the program, you begin to realize that you are printing one message, for instance the `"Turn printer on"` message, more often than any other message. It seems as if the `pr_msg()` function is receiving that message much more often than any other. This might be the case if you were writing a program that printed many reports to the printer. You still will use `pr_msg()` for other delayed messages, but the `"Turn printer on"` message is most frequently used.

Instead of calling the function over and over, typing the same message each time, you can set up the prototype for `pr_msg()` so it defaults to the `"Turn printer on"` in this way:

```
void pr_msg(char note[]="Turn printer on");// Prototype
```

List default argument values in the prototype.

After prototyping `pr_msg()` with the default argument list, C++ assumes you want to pass `"Turn printer on"` to the function unless you override the default by passing something else to it. For instance, in `main()`, you call `pr_msg()` this way:

```
pr_msg();    // C++ assumes you mean "Turn printer on".
```

This makes your programming job easier. Because most of the time you want `pr_msg()` to print `"Turn printer on"` the default

argument list takes care of the message and you do not have to pass the message when you call the function. However, those few times when you want to pass something else, simply pass a different message. For example, to make `pr_msg()` print `"Incorrect value"` you type:

```
pr_msg("Incorrect value");    // Pass a new message.
```

TIP: Any time you call a function several times and find yourself passing that function the same parameters most of the time, consider using a default argument list.

Multiple Default Arguments

You can specify more than one default argument in the prototype list. Here is a prototype for a function with three default arguments:

```
float funct1(int i=10, float x=7.5, char c='A');
```

There are several ways you can call this function. Here are some samples:

```
funct1();
```

All default values are assumed.

```
funct1(25);
```

A `25` is sent to the integer argument, and the default values are assumed for the rest.

```
funct1(25, 31.25);
```

A `25` is sent to the integer argument, `31.25` to the floating-point argument, and the default value of `'A'` is assumed for the character argument.

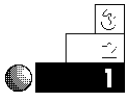
NOTE: If only some of a function's arguments are default arguments, those default arguments must appear on the far *left* of the argument list. No default arguments can appear to the left of those not specified as default. This is an *invalid* default argument prototype:

```
float func2(int i=10, float x, char c, long n=10.232);
```

This is invalid because a default argument appears on the left of a nondefault argument. To fix this, you have to move the two default arguments to the far left (the start) of the argument list. Therefore, by rearranging the prototype (and the resulting function calls) as follows, C++ enables you to accomplish the same objective as you attempted with the previous line:

```
float func2(float x, char c, int i=10, long n=10.232);
```

Examples



1. Here is a complete program that illustrates the message-printing function described earlier in this chapter. The `main()` function simply calls the delayed message-printing function three times, each time passing it a different set of argument lists.

```
// Filename: C20DEF1.CPP
// Illustrates default argument list.
#include <iostream.h>

void pr_msg(char note[]="Turn printer on"); // Prototype.

void main()
{
   pr_msg();                 // Prints default message.
   pr_msg("A new message");  // Prints another message.
   pr_msg();                 // Prints default message again.
   return;
}

void pr_msg(char note[])  // Only prototype contains defaults.
```

```
{
   long int delay;
   cout << note << "\n";
   for (delay=0; delay<500000; delay++)
      { ;  /* Do nothing while waiting */ }
   return;
}
```

The program produces the following output:

```
Turn printer on
A new message
Turn printer on
```

The delay loop causes each line to display for a couple of seconds or more, depending on the speed of your computer, until all three lines print.

2. The following program illustrates the use of defaulting several arguments. `main()` calls the function `de_fun()` five times, sending `de_fun()` five sets of arguments. The `de_fun()` function prints five different things depending on `main()`'s argument list.

```
// Filename: C20DEF2.CPP
// Demonstrates default argument list with several parameters.
#include <iostream.h>
#include <iomanip.h>

void de_fun(int i=5, long j=40034, float x=10.25,
            char ch='Z', double d=4.3234); // Prototype

void main()
{
   de_fun();          // All defaults used.
   de_fun(2);         // First default overridden.
   de_fun(2, 75037);  // First and second default overridden.
   de_fun(2, 75037, 35.88);      // First, second, and third
   de_fun(2, 75037, 35.88, 'G'); // First, second, third,
                                 // and fourth
   de_fun(2, 75037, 35.88, 'G', .0023); // No defaulting.
```

```
   return;
}

void de_fun(int i, long j, float x, char ch, double d)
{
   cout << setprecision(4) << "i: " << i << "    " << "j: " << j;
   cout << "    x: " << x << "    " << "ch: " << ch;
   cout << "    d: " << d << "\n";
   return;
}
```

Here is the output from this program:

```
i: 5    j: 40034    x: 10.25    ch: Z    d: 4.3234
i: 2    j: 40034    x: 10.25    ch: Z    d: 4.3234
i: 2    j: 75037    x: 10.25    ch: Z    d: 4.3234
i: 2    j: 75037    x: 35.88    ch: Z    d: 4.3234
i: 2    j: 75037    x: 35.88    ch: G    d: 4.3234
i: 2    j: 75037    x: 35.88    ch: G    d: 0.0023
```

Notice that each call to `de_fun()` produces a different output because `main()` sends a different set of parameters each time `main()` calls `de_fun()`.

Overloaded Functions

Unlike regular C, C++ enables you to have more than one function with the same name. In other words, you can have three functions called `abs()` in the same program. Functions with the same names are called overloaded functions. C++ requires that each overloaded function differ in its argument list. Overloaded functions enable you to have similar functions that work on different types of data.

For example, suppose you wrote a function that returned the absolute value of whatever number you passed to it. The absolute value of a number is its positive equivalent. For instance, the absolute value of 10.25 is 10.25 and the absolute value of –10.25 is 10.25.

Absolute values are used in distance, temperature, and weight calculations. The difference in the weights of two children is always

positive. If Joe weighs 65 pounds and Mary weighs 55 pounds, their difference is a positive 10 pounds. You can subtract the 65 from 55 (–10) or 55 from 65 (+10) and the weight difference is always the absolute value of the result.

Suppose you had to write an absolute-value function for integers, and an absolute-value function for floating-point numbers. Without function overloading, you need these two functions:

```
int iabs(int i)   // Returns absolute value of an integer.
{
   if (i < 0)
   {  return (i * -1); }  // Makes positive.
   else
   {  return (i);  }      // Already positive.
}

float fabs(float x)  // Returns absolute value of a float.
{
   if (x < 0.0)
   {  return (x * -1.0); }  // Makes positive.
   else
   {  return (x);  }        // Already positive.
}
```

Without overloading, if you had a floating-point variable for which you needed the absolute value, you pass it to the `fabs()` function as in:

```
ans = fabs(weight);
```

If you needed the absolute value of an integer variable, you pass it to the `iabs()` function as in:

```
ians = iabs(age);
```

Because the code for these two functions differ only in their parameter lists, they are perfect candidates for overloaded functions. Call both functions `abs()`, prototype both of them, and code each of them separately in your program. After overloading the two functions (each of which works on two different types of parameters with the same name), you pass your floating-point or integer value to `abs()`. The C++ compiler determines which function you wanted to call.

CAUTION: If two or more functions differ only in their return types, C++ cannot overload them. Two or more functions that differ only in their return types must have different names and cannot be overloaded.

This process simplifies your programming considerably. Instead of having to remember several different function names, you only have to remember one function name. C++ passes the arguments to the proper function.

NOTE: C++ uses *name-mangling* to accomplish overloaded functions. Understanding name-mangling helps you as you become an advanced C++ programmer.

When C++ realizes that you are overloading two or more functions with the same name, each function differing only in its parameter list, C++ changes the name of the function and adds letters to the end of the function name that match the parameters. Different C++ compilers do this differently.

To understand what the compiler does, take the absolute value function described earlier. C++ might change the integer absolute value function to `absi()` and the floating-point absolute value function to `absf()`. When you call the function with this function call:

```
ians = abs(age);
```

C++ determines that you want the `absi()` function called. As far as you know, C++ is not mangling the names; you never see the name differences in your program's source code. However, the compiler performs the name-mangling so it can keep track of different functions that have the same name.

Examples

1. Here is the complete absolute value program described in the previous text. Notice that both functions are prototyped. (The two prototypes signal C++ that it must perform name-mangling to determine the correct function names to call.)

```
// Filename: C20OVF1.CPP
// Overloads two absolute value functions.
#include <iostream.h>  // Prototype cout and cin.
#include <iomanip.h>   // Prototype setprecision(2).

int abs(int i);      // abs() is overloaded twice
float abs(float x);  // as shown by these prototypes.

void main()
{
   int ians;       // To hold return values.
   float fans;
   int i = -15;    // To pass to the two overloaded functions.
   float x = -64.53;

   ians = abs(i);    // C++ calls the integer abs().
   cout << "Integer absolute value of -15 is " << ians << "\n";

   fans = abs(x); // C++ calls the floating-point abs().
   cout << "Float absolute value of -64.53 is " <<
           setprecision(2) << fans << "\n";

   // Notice that you no longer have to keep track of two
   // different names. C++ calls the appropriate
   // function that matches the parameters.
   return;
}

int abs(int i)     // Integer absolute value function
{
   if (i < 0)
   {  return (i * -1); }  // Makes positive.
   else
   {  return (i);  }      // Already positive.
}
```

```
float abs(float x)  // Floating-point absolute value function
{
   if (x < 0.0)
   {  return (x * -1.0); }  // Makes positive.
   else
   {  return (x);  }        // Already positive.
}
```

The output from this program follows:

```
Integer absolute value of -15 is 15
Float absolute value of -64.53 is 64.53
```

2. As you write more and more C++ programs, you will see many uses for overloaded functions. The following program is a demonstration program showing how you can build your own output functions to suit your needs. `main()` calls three functions named `output()`. Each time it's called, `main()` passes a different value to the function.

   When `main()` passes `output()` a string, `output()` prints the string, formatted to a width (using the `setw()` manipulator described in Chapter 7, "Simple Input/Output") of 30 characters. When `main()` passes `output()` an integer, `output()` prints the integer with a width of five. When `main()` passes `output()` a floating-point value, `output()` prints the value to two decimal places and generalizes the output of different types of data. You do not have to format your own data. `output()` properly formats the data and you only have to remember one function name that outputs all three types of data.

```
// Filename: C20OVF2.CPP
// Outputs three different types of
// data with same function name.
#include <iostream.h>
#include <iomanip.h>
void output(char []);  // Prototypes for overloaded functions.
void output(int i);
void output(float x);
```

```
void main()
{
   char name[] = "C++ By Example makes C++ easy!";
   int ivalue = 2543;
   float fvalue = 39.4321;

   output(name);   // C++ chooses the appropriate function.
   output(ivalue);
   output(fvalue);

return;
}

void output(char name[])
{
   cout << setw(30) << name << "\n";
   // The width truncates string if it is longer than 30.
   return;
}

void output(int ivalue)
{
   cout << setw(5) << ivalue << "\n";
   // Just printed integer within a width of five spaces.
   return;
}

void output(float fvalue)
{
   cout << setprecision(2) << fvalue << "\n";
   // Limited the floating-point value to two decimal places.
   return;
}
```

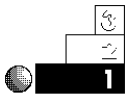
Here is the output from this program:

```
C++ By Example makes C++ easy!
2543
39.43
```

Each of the three lines, containing three different lines of information, was printed with the same function call.

Review Questions

The answers to the review questions are in Appendix B.



1. Where in the program do you specify the defaults for default argument lists?
2. What is the term for C++ functions that have the same name?

3. Does name-mangling help support default argument lists or overloaded functions?
4. True or false: You can specify only a single default argument.

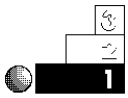
5. Fix the following prototype for a default argument list.

```
void my_fun(int i=7, float x, char ch='A');
```

6. True or false: The following prototypes specify overloaded functions:

```
int   sq_rt(int n);
float sq_rt(int n);
```

Review Exercises



1. Write a program that contains two functions. The first function returns the square of the integer passed to it, and the second function returns the square of the float passed to it.

2. Write a program that computes net pay based on the values the user types. Ask the user for the hours worked, the rate per hour, and the tax rate. Because the majority of employees work 40 hours per week and earn $5.00 per hour, use these values as default values in the function that computes the net pay. If the user presses Enter in response to your questions, use the default values.

Summary

Default argument lists and overloaded functions speed up your programming time. You no longer have to specify values for common arguments. You do not have to remember several different names for those functions that perform similar routines and differ only in their data types.

The remainder of this book elaborates on earlier concepts so you can take advantage of separate, modular functions and local data. You are ready to learn more about how C++ performs input and output. Chapter 21, "Device and Character Input/Output," teaches you the theory behind I/O in C++, and introduces more built-in functions.

Part V

Character Input/Output and String Functions

Device and Character Input/Output

Unlike many programming languages, C++ contains no input or output commands. C++ is an extremely *portable* language; a C++ program that compiles and runs on one computer is able also to compile and run on another type of computer. Most incompatibilities between computers reside in their input/output mechanics. Each different device requires a different method of performing I/O (Input/Output).

By putting all I/O capabilities in common functions supplied with each computer's compiler, not in C++ statements, the designers of C++ ensured that programs were not tied to specific hardware for input and output. A compiler has to be modified for every computer for which it is written. This ensures the compiler works with the specific computer and its devices. The compiler writers write I/O functions for each machine; when your C++ program writes a character to the screen, it works the same whether you have a color PC screen or a UNIX X/Windows terminal.

This chapter shows you additional ways to perform input and output of data besides the `cin` and `cout` functions you have seen

throughout the book. By providing character-based I/O functions, C++ gives you the basic I/O functions you need to write powerful data entry and printing routines.

This chapter introduces you to

- Stream input and output
- Redirecting I/O
- Printing to the printer
- Character I/O functions
- Buffered and nonbuffered I/O

By the time you finish this chapter, you will understand the fundamental built-in I/O functions available in C++. Performing character input and output, one character at a time, might sound like a slow method of I/O. You will soon realize that character I/O actually enables you to create more powerful I/O functions than `cin` and `cout`.

Stream and Character I/O

C++ views input and output from all devices as streams of characters.

C++ views all input and output as streams of characters. Whether your program receives input from the keyboard, a disk file, a modem, or a mouse, C++ only views a stream of characters. C++ does not have to know what type of device is supplying the input; the operating system handles the device specifics. The designers of C++ want your programs to operate on characters of data without regard to the physical method taking place.

This stream I/O means you can use the same functions to receive input from the keyboard as from the modem. You can use the same functions to write to a disk file, printer, or screen. Of course, you have to have some way of routing that stream input or output to the proper device, but each program's I/O functions works in a similar manner. Figure 21.1 illustrates this concept.

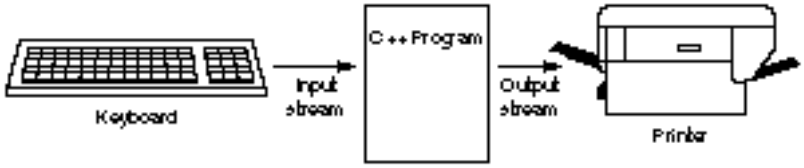


Figure 21.1. All I/O consists of streams of characters.

The Newline Special Character: `/n`

Portability is the key to C++'s success. Few companies have the resources to rewrite every program they use when they change computer equipment. They need a programming language that works on many platforms (hardware combinations). C++ achieves true portability better than almost any other programming language.

It is because of portability that C++ uses the generic newline character, `\n`, rather than the specific carriage return and line feed sequences other languages use. This is why C++ uses the `\t` for tab, as well as the other control characters used in I/O functions.

If C++ used ASCII code to represent these special characters, your programs would not be portable. You would write a C++ program on one computer and use a carriage return value such as 12, but 12 might not be the carriage return value on another type of computer.

By using newline and the other control characters available in C++, you ensure your program is compatible with any computer on which it is compiled. The specific compilers substitute their computer's actual codes for the control codes in your programs.

Standard Devices

Table 21.1 shows a listing of standard I/O devices. C++ always assumes input comes from *stdin*, meaning the *standard input device*. This is usually the keyboard, although you can reroute this default. C++ assumes all output goes to *stdout*, or the *standard output device*. There is nothing magic in the words stdin and stdout; however, many people learn their meanings for the first time in C++.

Table 21.1. Standard Devices in C++.

Description	*C++ Name*	*MS-DOS Name*
Screen	stdout	CON:
Keyboard	stdin	CON:
Printer	stdprn	PRN: or LPT1:
Serial Port	stdaux	AUX: or COM1:
Error Messages	stderr	CON:
Disk Files	none	Filename

Take a moment to study Table 21.1. You might think it is confusing that three devices are named CON:. MS-DOS differentiates between the screen device called CON: (which stands for *console*), and the keyboard device called CON: from the context of the data stream. If you send an output stream (a stream of characters) to CON:, MS-DOS routes it to the screen automatically. If you request input from CON:, MS-DOS retrieves the input from the keyboard. (These defaults hold true as long as you have not redirected these devices, as shown below.) MS-DOS sends all error messages to the screen (CON:) as well.

NOTE: If you want to route I/O to a second printer or serial port, see how to do so in Chapter 30, "Sequential Files."

Redirecting Devices from MS-DOS

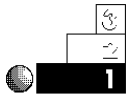
The operating system gives you control over devices.

The reason `cout` goes to the screen is simply because `stdout` is routed to the screen, by default, on most computers. The reason `cin` inputs from the keyboard is because most computers consider the keyboard to be the standard input device, stdin. After compiling your program, C++ does not send data to the screen or retrieve it from the keyboard. Instead, the program sends output to stdout and receives input from stdin. The operating system routes the data to the appropriate device.

MS-DOS enables you to reroute I/O from their default locations to other devices through the use of the *output redirection symbol*, >, and the *input redirection symbol*, <. The goal of this book is not to delve deeply in operating-system redirection. To learn more about the handling of I/O, read a good book on MS-DOS, such as *Using MS-DOS 5*.

Basically, the output redirection symbol informs the operating system that you want standard output to go to a device other than the default (the screen). The input redirection symbol routes input away from the keyboard to another input device. The following example illustrates how this is done in MS-DOS.

Examples



1. Suppose you write a program that uses only `cin` and `cout` for input and output. Instead of receiving input from the keyboard, you want the program to get the input from a file called MYDATA. Because `cin` receives input from `stdin`, you must redirect stdin. After compiling the program in a file called MYPGM.EXE, you can redirect its input away from the keyboard with the following DOS command:

   ```
   C:>MYPGM < MYDATA
   ```

   Of course, you can include a full pathname either before the program name or filename. There is a danger in redirecting all output such as this, however. All output, including screen prompts for keyboard input, goes to MYDATA. This is probably not acceptable to you in most cases; you still want

prompts and some messages to go to the screen. In the next section, you learn how to separate I/O, and send some output to one device such as the screen and the rest to another device, such as a file or printer.

2. You can also route the program's output to the printer by typing this:

```
C:>MYPGM > PRN:
```

Route MYPGM output to the printer.

3. If the program required much input, and that input were stored in a file called ANSWERS, you could override the keyboard default device that `cin` uses, as in:

```
C:>MYPGM < ANSWERS
```

The program reads from the file called ANSWERS every time `cin` *required input.*

4. You can combine redirection symbols. If you want input from the ANSWERS disk file, and want to send the output to the printer, do the following:

```
C:>MYPGM < ANSWERS > PRN:
```

TIP: You can route the output to a serial printer or a second parallel printer port by substituting COM1: or LPT2: for PRN:.

Printing Formatted Output to the Printer

`ofstream` allows your program to write to the printer.

It's easy to send program output to the printer using the `ofstream` function. The format of `ofstream` is

```
ofstream device(device_name);
```

ofstream uses the fstream.h header file.

The following examples show how you can combine cout and ofstream to write to both the screen and printer.

Example

The following program asks the user for his or her first and last name. It then prints the name, last name first, to the printer.

```
// Filename: C21FPR1.CPP
// Prints a name on the printer.

#include <fstream.h>

void main()
{
   char first[20];
   char last[20];

   cout << "What is your first name? ";
   cin >> first;
   cout << "What is your last name? ";
   cin >> last;

   // Send names to the printer.
   ofstream prn("PRN");
   prn << "In a phone book, your name looks like this: \n";
   prn << last << ", " << first << "\n";
   return;
}
```

Character I/O Functions

Because all I/O is actually character I/O, C++ provides many functions you can use that perform character input and output. The cout and cin functions are called *formatted I/O functions* because they give you formatting control over your input and output. The cout and cin functions are not character I/O functions.

There's nothing wrong with using `cout` for formatted output, but `cin` has many problems, as you have seen. You will now see how to write your own character input routines to replace `cin`, as well as use character output functions to prepare you for the upcoming section in this book on disk files.

The `get()` and `put()` Functions

`get()` and `put()` input and output characters from and to any standard devices.

The most fundamental character I/O functions are `get()` and `put()`. The `put()` function writes a single character to the standard output device (the screen if you don't redirect it from your operating system). The `get()` function inputs a single character from the standard input device (the keyboard by default).

The format for `get()` is

```
device.get(char_var);
```

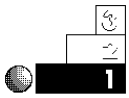
The `get()` *device* can be any standard input device. If you were receiving character input from the keyboard, you use `cin` as the device. If you initialize your modem and want to receive characters from it, use `ofstream` to open the modem device and read from the device.

The format of `put()` is

```
device.put(char_val);
```

The `char_val` can be a character variable, expression, or constant. You output character data with `put()`. The device can be any standard output device. To write a character to your printer, you open `PRN` with `ofstream`.

Examples



1. The following program asks the user for her or his initials a character at a time. Notice the program uses both `cout` and `put()`. The `cout` is still useful for formatted output such as messages to the user. Writing individual characters is best achieved with `put()`.

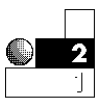
   The program has to call two `get()` functions for each character typed. When you answer a `get()` prompt by typing a

character followed by an Enter keypress, C++ interprets the input as a stream of two characters. The `get()` first receives the letter you typed, then it has to receive the `\n` (newline, supplied to C++ when you press Enter). There are examples that follow that fix this double `get()` problem.

```
// Filename: C21CH1.CPP
// Introduces get() and put().

#include <fstream.h>

void main()
{
   char  in_char;     // Holds incoming initial.
   char first, last;  // Holds converted first and last initial.

   cout << "What is your first name initial? ";
   cin.get(in_char);   // Waits for first initial.
   first = in_char;
   cin.get(in_char);   // Ignores newline.
   cout << "What is your last name initial? ";
   cin.get(in_char);   // Waits for last initial.
   last = in_char;
   cin.get(in_char);   // Ignores newline.
   cout << "\nHere they are: \n";
   cout.put(first);
   cout.put(last);
return;
}
```

Here is the output from this program:

```
What is your first name initial? G
What is your last name initial? P

Here they are:
GP
```



2. You can add carriage returns to space the output better. To print the two initials on two separate lines, use `put()` to put a newline character to `cout`, as the following program does:

```
// Filename: C21CH2.CPP
// Introduces get() and put() and uses put() to output
newline.

#include <fstream.h>

void main()
{
   char in_char;      // Holds incoming initial.
   char first, last;  // Holds converted first and last
                      // initial.

   cout << "What is your first name initial? ";
   cin.get(in_char);   // Waits for first initial.
   first = in_char;
   cin.get(in_char);   // Ignores newline.
   cout << "What is your last name initial? ";
   cin.get(in_char);   // Waits for last initial.
   last = in_char;
   cin.get(in_char);   // Ignores newline.
   cout << "\nHere they are: \n";
   cout.put(first);
   cout.put('\n');
   cout.put(last);
return;
}
```

3. It might have been clearer to define the newline character as a constant. At the top of the program, you have:

   ```
   const char NEWLINE='\n'
   ```

   The `put()` then reads:

   ```
   cout.put(NEWLINE);
   ```

Some programmers prefer to define their character formatting constants and refer to them by name. It's up to you to decide whether you want to use this method, or whether you want to continue using the `\n` character constant in `put()`.

The `get()` function is a *buffered* input function. As you type characters, the data does not immediately go to your program,

rather, it goes to a buffer. The buffer is a section of memory (and has nothing to do with your PC's type-ahead buffers) managed by C++.

Figure 21.2 shows how this buffered function works. When your program approaches a `get()`, the program temporarily waits as you type the input. The program doesn't view the characters, as they're going to the buffer of memory. There is practically no limit to the size of the buffer; it fills with input until you press Enter. Your Enter keypress signals the computer to release the buffer to your program.

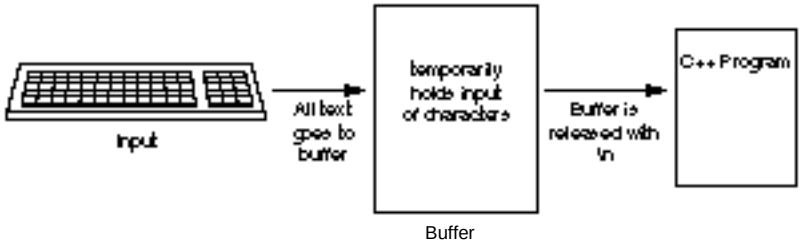


Figure 21.2. `get()` input goes to a buffer. The buffer is released when you press Enter.

Most PCs accept either buffered or nonbuffered input. The `getch()` function shown later in this chapter is nonbuffered. With `get()`, all input is buffered. Buffered text affects the timing of your program's input. Your program receives no characters from a `get()` until you press Enter. Therefore, if you ask a question such as

```
Do you want to see the report again (Y/N)?
```

and use `get()` for input, the user can press a `Y`, but the program does not receive the input until the user also presses Enter. The `Y` and Enter then are sent, one character at a time, to the program where it processes the input. If you want immediate response to a user's typing (such as the `INKEY$` in BASIC allows), you have to use `getch()`.

TIP: By using buffered input, the user can type a string of characters in response to a loop with `get()`, receive characters, and correct the input with Backspace before pressing Enter. If the input were nonbuffered, the Backspace would be another character of data.

Example

When receiving characters, you might have to discard the newline keypress.

C21CH2.CPP must discard the newline character. It did so by assigning the input character—from `get()`—to an extra variable. Obviously, the `get()` returns a value (the character typed). In this case, it's acceptable to ignore the return value by not using the character returned by `get()`. You know the user has to press Enter (to end the input) so it's acceptable to discard it with an unused `get()` function call.

When inputting strings such as names and sentences, `cin` only allows one word to be entered at a time. The following string asks the user for his or her full name with these two lines:

```
cout << "What are your first and last names? ";
cin >> names;        // Receive name in character array names.
```

The array `names` only receives the first name; `cin` ignores all data to the right of the first space.

You can build your own input function using `get()` that doesn't have a single-word limitation. When you want to receive a string of characters from the user, such as his or her first and last name, you can call the `get_in_str()` function shown in the next program.

The `main()` function defines an array and prompts the user for a name. After the prompt, the program calls the `get_in_str()` function and builds the input array a character at a time using `get()`. The function keeps looping, using the `while` loop, until the user presses Enter (signaled by the newline character, `\n`, to C++) or until the maximum number of characters are typed. You might want to use

this function in your own programs. Be sure to pass it a character array and an integer that holds the maximum array size (you don't want the input string to be longer than the character array that holds it). When control returns to `main()` (or whatever function called `get_in_str()`), the array has the user's full input, including the spaces.

```
// Filename: C21IN.CPP
// Program that builds an input string array using get().

#include <fstream.h>
void get_in_str(char str[], int len);

const int MAX=25;  // Size of character array to be typed.

void main()
{
   char input_str[MAX];   // Keyboard input fills this.
   cout << "What is your full name? ";
   get_in_str(input_str, MAX);   // String from keyboard
   cout << "After return, your name is " << input_str << "\n";
   return;
}

//*********************************************************
// The following function requires a string and the maximum
// length of the string be passed to it. It accepts input
// from the keyboard, and sends keyboard input in the string.
// On return, the calling routine has access to the string.
//*********************************************************

void get_in_str(char str[ ], int len)
{
   int i = 0;   // index
   char input_char;   // character typed

   cin.get(input_char);   // Get next character in string.
   while (i < (len - 1) && (input_char != '\n'))
     {
       str[i] = input_char;  // Build string a character
```

```
        i++;                    // at a time.
        cin.get(input_char);    // Receive next character in string.
      }
   str[i] = '\0';    // Make the char array a string.
   return;
}
```

NOTE: The loop checks for `len - 1` to save room for the null-terminating zero at the end of the input string.

The `getch()` and `putch()` Functions

The functions `getch()` and `putch()` are slightly different from the previous character I/O functions. Their format is similar to `get()` and `put()`; they read from the keyboard and write to the screen and cannot be redirected, even from the operating system. The formats of `getch()` and `putch()` are

```
int_var = getch();
```

and

```
putch(int_var);
```

`getch()` and `putch()` offer nonbuffered input and output that grab the user's characters immediately after the user types them.

`getch()` and `putch()` are not AT&T C++ standard functions, but they are usually available with most C++ compilers. `getch()` and `putch()` are nonbuffered functions. The `putch()` character output function is a mirror-image function to `getch()`; it is a nonbuffered output function. Because almost every output device made, except for the screen and modem, are inherently buffered, `putch()` effectively does the same thing as `put()`.

Another difference in `getch()` from the other character input functions is that `getch()` does not echo the input characters on the screen as it receives them. When you type characters in response to `get()`, you see the characters as you type them (as they are sent to the buffer). If you want to see characters received by `getch()`, you must follow `getch()` with a `putch()`. It is handy to echo the characters on the screen so the user can verify that she or he has typed correctly.

Characters input with `getch()` are not echoed to the screen as the user types them.

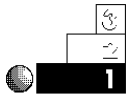
`getch()` and `putch()` use the conio.h header file.

Some programmers want to make the user press Enter after answering a prompt or selecting from a menu. They feel the extra time given with buffered input gives the user more time to decide if she or he wants to give that answer; the user can press Backspace and correct the input before pressing Enter.

Other programmers like to grab the user's response to a single-character answer, such as a menu response, and act on it immediately. They feel that pressing Enter is an added and unneeded burden for the user so they use `getch()`. The choice is yours. You should understand both buffered and nonbuffered input so you can use both.

TIP: You can also use `getche()`. `getche()` is a nonbuffered input identical to `getch()`, except the input characters are echoed (displayed) to the screen as the user types them. Using `getche()` rather than `getch()` keeps you from having to call a `putch()` to echo the user's input to the screen.

Example



The following program shows the `getch()` and `putch()` functions. The user is asked to enter five letters. These five letters are added (by way of a `for` loop) to the character array named `letters`. As you run this program, notice that the characters are not echoed to the screen as you type them. Because `getch()` is unbuffered, the program actually receives each character, adds it to the array, and loops again, as you type them. (If this were buffered input, the program would not loop through the five iterations until you pressed Enter.)

A second loop prints the five letters using `putch()`. A third loop prints the five letters to the printer using `put()`.

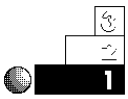
```
// Filename: C21GCH1.CPP
// Uses getch() and putch() for input and output.

#include <fstream.h>
```

```
#include <conio.h>

void main()
{
   int ctr;   // for loop counter
   char letters[5];   // Holds five input characters. No
                      // room is needed for the null zero
                      // because this array never will be
                      // treated as a string.
   cout << "Please type five letters... \n";
   for (ctr = 0; ctr < 5; ctr++)
     {
       letters[ctr] = getch();     // Add input to array.
     }
   for (ctr = 0; ctr < 5; ctr++)   // Print them to screen.
   {
      putch(letters[ ctr ]);
   }
   ofstream prn("PRN");
   for (ctr = 0; ctr < 5; ctr++)   // Print them to printer.
   {
      prn.put(letters[ ctr ]);
   }
return;
}
```

When you run this program, do not press Enter after the five letters. The `getch()` function does not use the Enter. The loop automatically ends after the fifth letter because of the unbuffered input and the `for` loop.

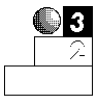
Review Questions

The answers to the review questions are found in Appendix B.



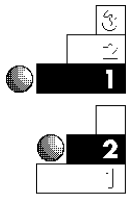
1. Why are there no input or output commands in C++?
2. True or false: If you use the character I/O functions to send output to stdout, it always goes to the screen.

3. What is the difference between `getch()` and `get()`?
4. What function sends formatted output to devices other than the screen?
5. What are the MS-DOS redirection symbols?
6. What nonstandard function, most similar to `getch()`, echoes the input character to the screen as the user types it?



7. True or false: When using `get()`, the program receives your input as you type it.
8. Which keypress releases the buffered input to the program?
9. True or false: Using devices and functions described in this chapter, it is possible to write one program that sends some output to the screen, some to the printer, and some to the modem.

Review Exercises



1. Write a program that asks the user for five letters and prints them in reverse order to the screen, and then to the printer.
2. Write a miniature typewriter program, using `get()` and `put()`. In a loop, get characters until the user presses Enter while he or she is getting a line of user input. Write the line of user input to the printer. Because `get()` is buffered, nothing goes to the printer until the user presses Enter at the end of each line of text. (Use the string-building input function shown in C21IN.CPP.)
3. Add a `putch()` inside the first loop of C21CH1.CPP (this chapter's first `get()` example program) so the characters are echoed to the screen as the user types them.

4. A *palindrome* is a word or phrase spelled the same forwards and backwards. Two example palindromes are

```
Madam, I'm Adam
Golf? No sir, prefer prison flog!
```

Write a C++ program that asks the user for a phrase. Build the input, a character at a time, using a character input function such as `get()`. Once you have the full string (store it in a character array), determine whether the phrase is a palindrome. You have to filter special characters (nonalphabetic), storing only alphabetic characters to a second character array. You also must convert the characters to uppercase as you store them. The first palindrome becomes:

```
MADAMIMADAM
```

Using one or more `for` or `while` loops, you can now test the phrase to determine whether it is a palindrome. Print the result of the test on the printer. Sample output should look like:

```
"Madam, I'm Adam" is a palindrome.
```

Summary

You now should understand the generic methods C++ programs use for input and output. By writing to standard I/O devices, C++ achieves portability. If you write a program for one computer, it works on another. If C++ were to write directly to specific hardware, programs would not work on every computer.

If you still want to use the formatted I/O functions, such as `cout`, you can do so. The `ofstream()` function enables you to write formatted output to any device, including the printer.

The methods of character I/O might seem primitive, and they are, but they give you the flexibility to build and create your own input functions. One of the most often-used C++ functions, a string-building character I/O function, was demonstrated in this chapter (the C21IN.CPP program).

The next two chapters (Chapter 22, "Character, String, and Numeric Functions," and Chapter 23, "Introducing Arrays") introduce many character and string functions, including string I/O functions. The string I/O functions build on the principles presented here. You will be surprised at the extensive character and string manipulation functions available in the language as well.

Character, String, and Numeric Functions

C++ provides many built-in functions in addition to the `cout`, `getch()`, and `strcpy()` functions you have seen so far. These built-in functions increase your productivity and save you programming time. You don't have to write as much code because the built-in functions perform many useful tasks for you.

This chapter introduces you to

- Character conversion functions
- Character and string testing functions
- String manipulation functions
- String I/O functions
- Mathematical, trigonometric, and logarithmic functions
- Random-number processing

Character Functions

This section explores many of the character functions available in AT&T C++. Generally, you pass character arguments to the functions, and the functions return values that you can store or print. By using these functions, you off-load much of your work to C++ and allow it to perform the more tedious manipulations of character and string data.

Character Testing Functions

The character functions return True or False results based on the characters you pass to them.

Several functions test for certain characteristics of your character data. You can determine whether your character data is alphabetic, digital, uppercase, lowercase, and much more. You must pass a character variable or literal argument to the function (by placing the argument in the function parentheses) when you call it. These functions return a True or False result, so you can test their return values inside an `if` statement or a `while` loop.

NOTE: All character functions presented in this section are prototyped in the ctype.h header file. Be sure to include ctype.h at the beginning of any programs that use them.

Alphabetic and Digital Testing

The following functions test for alphabetic conditions:

- `isalpha(c)`: Returns True (nonzero) if `c` is an uppercase or lowercase letter. Returns False (zero) if `c` is not a letter.
- `islower(c)`: Returns True (nonzero) if `c` is a lowercase letter. Returns False (zero) if `c` is not a lowercase letter.
- `isupper(c)`: Returns True (nonzero) if `c` is an uppercase letter. Returns False (zero) if `c` is not an uppercase letter.

Remember that any nonzero value is True in C++, and zero is always False. If you use the return values of these functions in a relational test, the True return value is not always 1 (it can be any nonzero value), but it is always considered True for the test.

The following functions test for digits:

- `isdigit(c)`: Returns True (nonzero) if `c` is a digit 0 through 9. Returns False (zero) if `c` is not a digit.
- `isxdigit(c)`: Returns True (nonzero) if `c` is any of the hexadecimal digits 0 through 9 or A, B, C, D, E, F, a, b, c, d, e, or f. Returns False (zero) if `c` is anything else. (See Appendix A, "Memory Addressing, Binary, and Hexadecimal Review," for more information on the hexadecimal numbering system.)

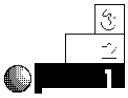
NOTE: Although some character functions test for digits, the arguments are still character data and cannot be used in mathematical calculations, unless you calculate using the ASCII values of characters.

The following function tests for numeric or alphabetical arguments:

`isalnum(c)`: Returns True (nonzero) if `c` is a digit 0 through 9 or an alphabetic character (either uppercase or lowercase). Returns False (zero) if `c` is not a digit or a letter.

CAUTION: You can pass to these functions only a character value or an integer value holding the ASCII value of a character. You cannot pass an entire character array to character functions. If you want to test the elements of a character array, you must pass the array one element at a time.

Example



The following program asks users for their initials. If a user types anything but alphabetic characters, the program displays an error and asks again.

Identify the program and include the input/output header files. The program asks the user for his or her first initial, so declare the character variable `initial` *to hold the user's answer.*

1. *Ask the user for her or his first initial, and retrieve the user's answer.*
2. *If the answer was not an alphabetic character, tell the user this and repeat step one.*

Print a thank-you message on-screen.

```
// Filename: C22INI.CPP
// Asks for first initial and tests
// to ensure that response is correct.
#include <iostream.h>
#include <ctype.h>
void main()
{
   char initial;
   cout << "What is your first initial? ";
   cin >> initial;

   while (!isalpha(initial))
     {
       cout << "\nThat was not a valid initial! \n";
       cout << "\nWhat is your first initial? ";
       cin >> initial;
     }

   cout << "\nThanks!";
   return;
}
```

This use of the `not` operator (`!`) is quite clear. The program continues to loop as long as the entered character is not alphabetic.

Special Character-Testing Functions

A few character functions become useful when you have to read from a disk file, a modem, or another operating system device that you route input from. These functions are not used as much as the character functions you saw in the previous section, but they are useful for testing specific characters for readability.

The character-testing functions do not change characters.

The remaining character-testing functions follow:

- `iscntrl(c)`: Returns True (nonzero) if `c` is a *control character* (any character from the ASCII table numbered 0 through 31). Returns False (zero) if `c` is not a control character.
- `isgraph(c)`: Returns True (nonzero) if `c` is any printable character (a noncontrol character) except a space. Returns False (zero) if `c` is a space or anything other than a printable character.
- `isprint(c)`: Returns True (nonzero) if `c` is a printable character (a noncontrol character) from ASCII 32 to ASCII 127, including a space. Returns False (zero) if `c` is not a printable character.
- `ispunct(c)`: Returns True (nonzero) if `c` is any punctuation character (any printable character other than a space, a letter, or a digit). Returns False (zero) if `c` is not a punctuation character.
- `isspace(c)`: Returns True (nonzero) if `c` is a space, newline (`\n`), carriage return (`\r`), tab (`\t`), or vertical tab (`\v`) character. Returns False (zero) if `c` is anything else.

Character Conversion Functions

Both `tolower()` and `toupper()` return lowercase or uppercase arguments.

The two remaining character functions are handy. Rather than test characters, these functions change characters to their lower- or uppercase equivalents.

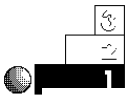
- `tolower(c)`: Converts `c` to lowercase. Nothing changes if you pass `tolower()` a lowercase letter or a nonalphabetic character.
- `toupper(c)`: Converts `c` to uppercase. Nothing changes if you pass `toupper()` an uppercase letter or a nonalphabetic character.

These functions return their changed character values. These functions are useful for user input. Suppose you are asking users a yes or no question, such as the following:

```
Do you want to print the checks (Y/N)?
```

Before `toupper()` and `tolower()` were developed, you had to check for both a *Y* and a *y* to print the checks. Instead of testing for both conditions, you can convert the character to uppercase, and test for a *Y*.

Example



Here is a program that prints an appropriate message if the user is a girl or a boy. The program tests for *G* and *B* after converting the user's input to uppercase. No check for lowercase has to be done.

Identify the program and include the input/output header files. The program asks the user a question requiring an alphabetic answer, so declare the character variable `ans` *to hold the user's response.*

Ask whether the user is a girl or a boy, and store the user's answer in `ans`. *The user must press Enter, so incorporate and then discard the Enter keypress. Change the value of* `ans` *to uppercase. If the answer is G, print a message. If the answer is B, print a different message. If the answer is something else, print another message.*

```
// Filename: C22GB.CPP
// Determines whether the user typed a G or a B.
#include <iostream.h>
#include <conio.h>
#include <ctype.h>
void main()
{
```

```
   char ans;                    // Holds user's response.
   cout << "Are you a girl or a boy (G/B)? ";
   ans=getch();                              // Get answer.
   getch();                            // Discard newline.

cout <<ans<<"\n";
   ans = toupper(ans);     // Convert answer to uppercase.
   switch (ans)
   {   case ('G'): { cout << "You look pretty today!\n";
                     break; }
       case ('B'): { cout << "You look handsome today!\n";
                     break; }
       default :   { cout << "Your answer makes no sense!\n";
                     break; }
   }
   return;
}
```

Here is the output from the program:

```
Are you a girl or a boy (G/B)? B
You look handsome today!
```

String Functions

Some of the most powerful built-in C++ functions are the string functions. They perform much of the tedious work for which you have been writing code so far, such as inputting strings from the keyboard and comparing strings.

As with the character functions, there is no need to "reinvent the wheel" by writing code when built-in functions do the same task. Use these functions as much as possible.

Now that you have a good grasp of the foundations of C++, you can master the string functions. They enable you to concentrate on your program's primary purpose, rather than spend time coding your own string functions.

Useful String Functions

You can use a handful of useful string functions for string testing and conversion. You have already seen (in earlier chapters) the `strcpy()` string function, which copies a string of characters to a character array.

NOTE: All string functions in this section are prototyped in the string.h header file. Be sure to include string.h at the beginning of any program that uses the string functions.

The string functions work on string literals or on character arrays that contain strings.

String functions that test or manipulate strings follow:

- `strcat(s1, s2)`: Concatenates (merges) the `s2` string to the end of the `s1` character array. The `s1` array must have enough reserved elements to hold both strings.
- `strcmp(s1, s2)`: Compares the `s1` string with the `s2` string on an alphabetical, element-by-element basis. If `s1` alphabetizes before `s2`, `strcmp()` returns a negative value. If `s1` and `s2` are the same strings, `strcmp()` returns `0`. If `s1` alphabetizes after `s2`, `strcmp()` returns a positive value.
- `strlen(s1)`: Returns the length of `s1`. Remember, the length of a string is the number of characters, not including the null zero. The number of characters defined for the character array has nothing to do with the length of the string.

TIP: Before using `strcat()` to concatenate strings, use `strlen()` to ensure that the target string (the string being concatenated to) is large enough to hold both strings.

String I/O Functions

In the previous few chapters, you have used a character input function, `cin.get()`, to build input strings. Now you can begin to use the string input and output functions. Although the goal of the

string-building functions has been to teach you the specifics of the language, these string I/O functions are much easier to use than writing a character input function.

The string input and output functions are listed as follows:

- `gets(s)`: Stores input from `stdin` (usually directed to the keyboard) to the string named `s`.
- `puts(s)`: Outputs the `s` string to `stdout` (usually directed to the screen by the operating system).
- `fgets(s, len, dev)`: Stores input from the standard device specified by `dev` (such as `stdin` or `stdaux`) in the `s` string. If more than `len` characters are input, `fgets()` discards them.
- `fputs(s, dev)`: Outputs the `s` string to the standard device specified by `dev`.

Both `gets()` and `puts()` input and output strings.

These four functions make the input and output of strings easy. They work in pairs. That is, strings input with `gets()` are usually output with `puts()`. Strings input with `fgets()` are usually output with `fputs()`.

TIP: `gets()` replaces the string-building input function you saw in earlier chapters.

Terminate `gets()` or `fgets()` input by pressing Enter. Each of these functions handles string-terminating characters in a slightly different manner, as follows:

`gets()`	A newline input becomes a null zero (`\0`).
`puts()`	A null at the end of the string becomes a newline character (`\n`).
`fgets()`	A newline input stays, and a null zero is added after it.
`fputs()`	The null zero is dropped, and a newline character is not added.

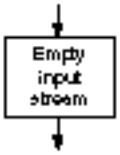
Therefore, when you enter strings with `gets()`, C++ places a string-terminating character in the string at the point where you press Enter. This creates the input string. (Without the null zero, the

input would not be a string.) When you output a string, the null zero at the end of the string becomes a newline character. This is preferred because a newline is at the end of a line of output and the cursor begins automatically on the next line.

Because `fgets()` and `fputs()` can input and output strings from devices such as disk files and telephone modems, it can be critical that the incoming newline characters are retained for the data's integrity. When outputting strings to these devices, you do not want C++ inserting extra newline characters.

CAUTION: Neither `gets()` nor `fgets()` ensures that its input strings are large enough to hold the incoming data. It is up to you to make sure enough space is reserved in the character array to hold the complete input.

One final function is worth noting, although it is not a string function. It is the `fflush()` function, which flushes (empties) whatever standard device is listed in its parentheses. To flush the keyboard of all its input, you would code as follows:



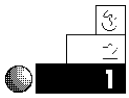

```
fflush(stdin);
```

When you are doing string input and output, sometimes an extra newline character appears in the keyboard buffer. A previous answer to `gets()` or `getc()` might have an extra newline you forgot to discard. When a program seems to ignore `gets()`, you might have to insert `fflush(stdin)` before `gets()`.

Flushing the standard input device causes no harm, and using it can clear the input stream so your next `gets()` works properly. You can also flush standard output devices with `fflush()` to clear the output stream of any characters you sent to it.

NOTE: The header file for `fflush()` is in stdio.h.

Example



The following program shows you how easy it is to use `gets()` and `puts()`. The program requests the name of a book from the user using a single `gets()` function call, then prints the book title with `puts()`.

Identify the program and include the input/output header files. The program asks the user for the name of a book. Declare the character array `book` *with 30 elements to hold the user's answer.*

Ask the user for the book's title, and store the user's response in the `book` *array. Display the string stored in* `book` *to an output device, probably your screen. Print a thank-you message.*

```
// C22GPS1.CPP
// Inputs and outputs strings.
#include <iostream.h>
#include <stdio.h>
#include <string.h>

void main()
{
   char book[30];

   cout << "What is the book title? ";
   gets(book);                    // Get an input string.
   puts(book);                    // Display the string.
   cout << "Thanks for the book!\n";
   return;
}
```

The output of the program follows:

```
What is the book title? Mary and Her Lambs
Mary and Her Lambs
Thanks for the book!
```

Converting Strings to Numbers

Sometimes you have to convert numbers stored in character strings to a numeric data type. AT&T C++ provides three functions that enable you to do this:

- `atoi(s)`: Converts `s` to an integer. The name stands for *a*lphabetic *to i*nteger.
- `atol(s)`: Converts `s` to a long integer. The name stands for *a*lphabetic *to l*ong integer.
- `atof(s)`: Converts `s` to a floating-point number. The name stands for *a*lphabetic *to f*loating-point.

NOTE: These three `ato()` functions are prototyped in the stdlib.h header file. Be sure to include stdlib.h at the beginning of any program that uses the `ato()` functions.

The string must contain a valid number. Here is a string that can be converted to an integer:

```
"1232"
```

The string must hold a string of digits short enough to fit in the target numeric data type. The following string could not be converted to an integer with the `atoi()` function:

```
"-1232495.654"
```

However, it could be converted to a floating-point number with the `atof()` function.

C++ cannot perform any mathematical calculation with such strings, even if the strings contain digits that represent numbers. Therefore, you must convert any string to its numeric equivalent before performing arithmetic with it.

NOTE: If you pass a string to an `ato()` function and the string does not contain a valid representation of a number, the `ato()` function returns `0`.

These functions become more useful to you after you learn about disk files and pointers.

Numeric Functions

This section presents many of the built-in C++ numeric functions. As with the string functions, these functions save you time by converting and calculating numbers instead of your having to write functions that do the same thing. Many of these are trigonometric and advanced mathematical functions. You might use some of these numeric functions only rarely, but they are there if you need them.

This section concludes the discussion of C++'s standard built-in functions. After mastering the concepts in this chapter, you are ready to learn more about arrays and pointers. As you develop more skills in C++, you might find yourself relying on these numeric, string, and character functions when you write more powerful programs.

Useful Mathematical Functions

Several built-in numeric functions return results based on numeric variables and literals passed to them. Even if you write only a few science and engineering programs, some of these functions are useful.

NOTE: All mathematical and trigonometric functions are prototyped in the math.h header file. Be sure to include math.h at the beginning of any program that uses the numeric functions.

These numeric functions return double-precision values.

Here are the functions listed with their descriptions:

- `ceil(x)`: The `ceil()`, or *ceiling*, function rounds numbers up to the nearest integer.
- `fabs(x)`: Returns the absolute value of `x`. The absolute value of a number is its positive equivalent.

TIP: Absolute value is used for distances (which are always positive), accuracy measurements, age differences, and other calculations that require a positive result.

- `floor(x)`: The `floor()` function rounds numbers down to the nearest integer.
- `fmod(x, y)`: The `fmod()` function returns the floating-point remainder of (`x/y`) with the same sign as `x`, and `y` cannot be zero. Because the modulus operator (`%`) works only with integers, this function is used to find the remainder of floating-point number divisions.
- `pow(x, y)`: Returns `x` raised to the `y` power, or x^y. If `x` is less than or equal to zero, `y` must be an integer. If `x` equals zero, `y` cannot be negative.
- `sqrt(x)`: Returns the square root of `x`; `x` must be greater than or equal to zero.

The *n*th Root

No function returns the *n*th root of a number, only the square root. In other words, you cannot call a function that gives you the 4th root of 65,536. (By the way, 16 is the 4th root of 65,536, because 16 times 16 times 16 times 16 = 65,536.)

You can use a mathematical trick to simulate the *n*th root, however. Because C++ enables you to raise a number to a fractional power—with the `pow()` function—you can raise a number to the *n*th root by raising it to the (1/*n*) power. For example, to find the 4th root of 65,536, you could type this:

```
root = pow(65536.0, (1.0/4.0));
```

Note that the decimal point keeps the numbers in floating-point format. If you leave them as integers, such as

```
root = pow(65536, (1/4));
```

C++ produces incorrect results. The `pow()` function and most other mathematical functions require floating-point values as arguments.

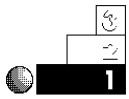
To store the 7th root of 78,125 in a variable called `root`, for example, you would type

```
root = pow(78125.0, (1.0/7.0));
```

This stores 5.0 in `root` because 5^7 equals 78,125.

Knowing how to compute the nth root is handy in scientific programs and also in financial applications, such as time-value-of-money problems.

Example



The following program uses the `fabs()` function to compute the difference between two ages.

```
// Filename: C22ABS.CPP
// Computes the difference between two ages.
#include <iostream.h>
#include <math.h>
void main()
{
   float age1, age2, diff;
   cout << "\nWhat is the first child's age? ";
   cin >> age1;
   cout << "What is the second child's age? ";
   cin >> age2;

   // Calculates the positive difference.
   diff = age1 - age2;
   diff = fabs(diff);     // Determines the absolute value.

   cout << "\nThey are " << diff << " years apart.";
   return;
}
```

The output from this program follows. Due to `fabs()`, the order of the ages doesn't matter. Without absolute value, this program would produce a negative age difference if the first age was less than the second. Because the ages are relatively small, floating-point variables are used in this example. C++ automatically converts floating-point arguments to double precision when passing them to `fabs()`.

```
What is the first child's age? 10
What is the second child's age? 12

They are 2 years apart.
```

Trigonometric Functions

The following functions are available for trigonometric applications:

- `cos(x)`: Returns the cosine of the angle `x`, expressed in radians.
- `sin(x)`: Returns the sine of the angle `x`, expressed in radians.
- `tan(x)`: Returns the tangent of the angle `x`, expressed in radians.

These are probably the least-used functions. This is not to belittle the work of scientific and mathematical programmers who need them, however. Certainly, they are grateful that C++ supplies these functions! Otherwise, programmers would have to write their own functions to perform these three basic trigonometric calculations.

Most C++ compilers supply additional trigonometric functions, including hyperbolic equivalents of these three functions.

TIP: If you have to pass an angle that is expressed in degrees to these functions, convert the angle's degrees to radians by multiplying the degrees by $\pi/180.0$ (π equals approximately 3.14159).

Logarithmic Functions

Three highly mathematical functions are sometimes used in business and mathematics. They are listed as follows:

- `exp(x)`: Returns the base of natural logarithm (*e*) raised to a power specified by `x` (e^x); *e* is the mathematical expression for the approximate value of 2.718282.
- `log(x)`: Returns the natural logarithm of the argument `x`, mathematically written as `ln(x)`. `x` must be positive.
- `log10(x)`: Returns the base-10 logarithm of argument `x`, mathematically written as `log10(x)`. `x` must be positive.

Random-Number Processing

Random events happen every day. You wake up and it is sunny or rainy. You have a good day or a bad day. You get a phone call from an old friend or you don't. Your stock portfolio might go up or down in value.

Random events are especially important in games. Part of the fun in games is your luck with rolling dice or drawing cards, combined with your playing skills.

Simulating random events is an important task for computers. Computers, however, are finite machines; given the same input, they always produce the same output. This fact can create some boring games!

The `rand()` function produces random integer numbers.

The designers of C++ knew this computer setback and found a way to overcome it. They wrote a random-number generating function called `rand()`. You can use `rand()` to compute a dice roll or draw a card, for example.

To call the `rand()` function and assign the returned random number to test, use the following syntax:

```
test = rand();
```

The `rand()` function returns an integer from 0 to 32,767. Never use an argument in the `rand()` parentheses.

Every time you call `rand()` in the same program, you receive a different number. If you run the same program over and over,

however, rand() returns the same set of random numbers. One way to receive a different set of random numbers is to call the srand() function. The format of srand() follows:

```
srand(seed);
```

where seed is an integer variable or literal. If you don't call srand(), C++ assumes a seed value of 1.

NOTE: The rand() and srand() functions are prototyped in the stdlib.h header file. Be sure to include stdlib.h at the beginning of any program that uses rand() or srand().

The seed value reseeds, or resets, the random-number generator, so the next random number is based on the new seed value. If you call srand() with a different seed value at the top of a program, rand() returns a different random number each time you run the program.

Why Do You Have To Do This?

There is considerable debate among C++ programmers concerning the random-number generator. Many think that the random numbers should be truly random, and that they should not have to seed the generator themselves. They think that C++ should do its own internal seeding when you ask for a random number.

However, many applications would no longer work if the random-number generator were randomized for you. Computers are used in business, engineering, and research to simulate the pattern of real-world events. Researchers have to be able to duplicate these simulations, over and over. Even though the events inside the simulations might be random from each other, the running of the simulations cannot be random if researchers are to study several different effects.

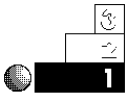
Mathematicians and statisticians also have to repeat random-number patterns for their analyses, especially when they work with risk, probability, and gaming theories.

Because so many computer users have to repeat their random-number patterns, the designers of C++ have wisely chosen to give you, the programmer, the option of keeping the same random patterns or changing them. The advantages far outweigh the disadvantage of including an extra `srand()` function call.

If you want to produce a different set of random numbers every time your program runs, you must determine how your C++ compiler reads the computer's system clock. You can use the seconds count from the clock to seed the random-number generator so it seems truly random.

Review Questions

The answers to the review questions are in Appendix B.



1. How do the character testing functions differ from the character conversion functions?
2. What are the two string input functions?
3. What is the difference between `floor()` and `ceil()`?

4. What does the following nested function return?

```
isalpha(islower('s'));
```

5. If the character array `str1` contains the string `Peter` and the character array `str2` contains `Parker`, what does `str2` contain after the following line of code executes?

```
strcat(str1, str2);
```

6. What is the output of the following `cout`?

```
cout << floor(8.5) << " " << ceil(8.5);
```

7. True or false: The `isxdigit()` and `isgraph()` functions could return the same value, depending on the character passed to them.

8. Assume you declare a character array with the following statement:

   ```
   char ara[5];
   ```

   Now suppose the user types `Programming` in response to the following statement:

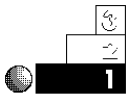
   ```
   fgets(ara, 5, stdin);
   ```

   Would `ara` contain `Prog`, `Progr`, or `Programming`?

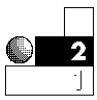
9. True or false: The following statements print the same results.

   ```
   cout << pow(64.0, (1.0/2.0)) ;
   cout << sqrt(64.0);
   ```

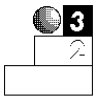
Review Exercises



1. Write a program that asks users for their ages. If a user types anything other than two digits, display an error message.
2. Write a program that stores a password in a character array called `pass`. Ask users for the password. Use `strcmp()` to inform users whether they typed the proper password. Use the string I/O functions for all the program's input and output.
3. Write a program that rounds up and rounds down the numbers –10.5, –5.75, and 2.75.



4. Ask users for their names. Print every name in *reverse* case; print the first letter of each name in lowercase and the rest of the name in uppercase.
5. Write a program that asks users for five movie titles. Print the longest title. Use only the string I/O and manipulation functions presented in this chapter.
6. Write a program that computes the square root, cube root, and fourth root of the numbers from 10 to 25, inclusive.



7. Ask users for the titles of their favorite songs. Discard all the special characters in each title. Print the words in the title, one per line. For example, if they enter `My True Love Is Mine, Oh, Mine!`, you should output the following:

```
My
True
Love
Is
Mine
Oh
Mine
```

8. Ask users for the first names of 10 children. Using `strcmp()` on each name, write a program to print the name that comes first in the alphabet.

Summary

You have learned the character, string, and numeric functions that C++ provides. By including the ctype.h header file, you can test and convert characters that a user types. These functions have many useful purposes, such as converting a user's response to uppercase. This makes it easier for you to test user input.

The string I/O functions give you more control over both string and numeric input. You can receive a string of digits from the keyboard and convert them to a number with the `ato()` functions. The string comparison and concatenation functions enable you to test and change the contents of more than one string.

Functions save you programming time because they take over some of your computing tasks, leaving you free to concentrate on your programs. C++'s numeric functions round and manipulate numbers, produce trigonometric and logarithmic results, and produce random numbers.

Now that you have learned most of C++'s built-in functions, you are ready to improve your ability to work with arrays. Chapter 23, "Introducing Arrays," extends your knowledge of character arrays and shows you how to produce arrays of any data type.