Part Ⅴ Character InputOutput and String Functions(五)

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.

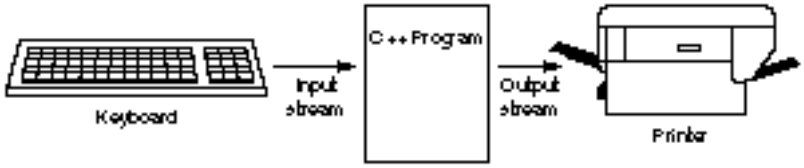


Figure 21.1. All I/O consists of streams of characters.

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

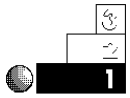
The operating system gives you control over devices.

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);
```

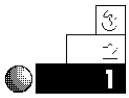
The `get()` *device* can be any standard input device. If you were receiving character input from the keyboard, you use `cin` as the device. If you initialize your modem and want to receive characters from it, use `ofstream` to open the modem device and read from the device.

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()`.

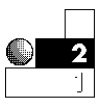
   The program has to call two `get()` functions for each character typed. When you answer a `get()` prompt by typing a

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.

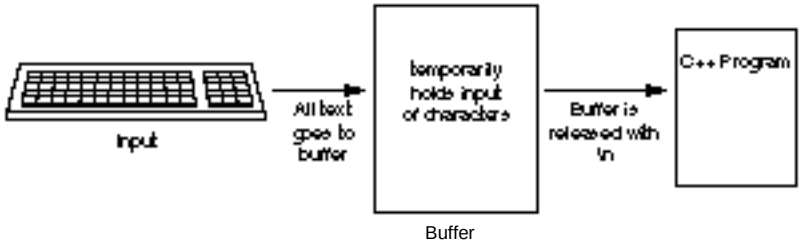


Figure 21.2. `get()` input goes to a buffer. The buffer is released when you press Enter.

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.

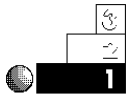
`getch()` and `putch()` use the conio.h header file.

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()`.

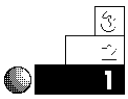
```
// Filename: C21GCH1.CPP
// Uses getch() and putch() for input and output.

#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.

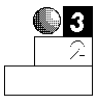
Review Questions

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



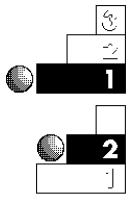
1. Why are there no input or output commands in C++?
2. True or false: If you use the character I/O functions to send output to stdout, it always goes to the screen.

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.)

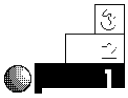
NOTE: Although some character functions test for digits, the arguments are still character data and cannot be used in mathematical calculations, unless you calculate using the ASCII values of characters.

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.

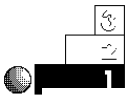
- `tolower(c)`: Converts `c` to lowercase. Nothing changes if you pass `tolower()` a lowercase letter or a nonalphabetic character.
- `toupper(c)`: Converts `c` to uppercase. Nothing changes if you pass `toupper()` an uppercase letter or a nonalphabetic character.

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.

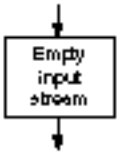
Therefore, when you enter strings with `gets()`, C++ places a string-terminating character in the string at the point where you press Enter. This creates the input string. (Without the null zero, the

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:



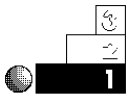

```
fflush(stdin);
```

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.

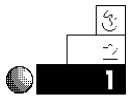
To store the 7th root of 78,125 in a variable called `root`, for example, you would type

```
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.

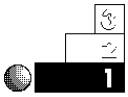
Mathematicians and statisticians also have to repeat random-number patterns for their analyses, especially when they work with risk, probability, and gaming theories.

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:

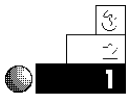
   ```
   fgets(ara, 5, stdin);
   ```

   Would `ara` contain `Prog`, `Progr`, or `Programming`?

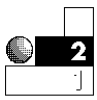
9. True or false: The following statements print the same results.

   ```
   cout << pow(64.0, (1.0/2.0)) ;
   cout << sqrt(64.0);
   ```

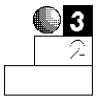
Review Exercises



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.