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.

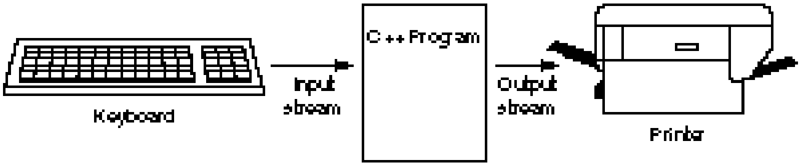

Figure 21.1. All I/O consists of streams of characters.

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.

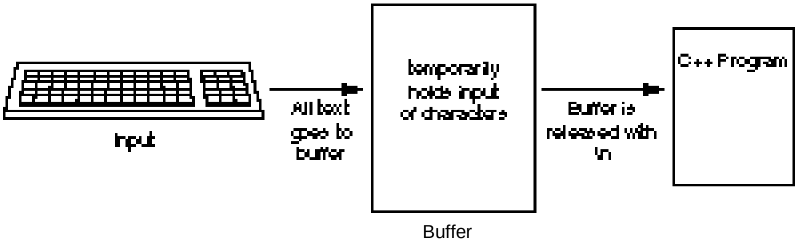
Input
All text goes to buffer
temporarily holds input of characters
Buffer
Buffer is released with \n
C++ Program

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()` 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()`.

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

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

1

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.