Simple Input/Output

You have already seen the `cout` operator. It prints values to the screen. There is much more to `cout` than you have learned. Using `cout` and the screen (the most common output device), you can print information any way you want it. Your programs also become much more powerful if you learn to receive input from the keyboard. `cin` is an operator that mirrors the `cout`. Instead of sending output values to the screen, `cin` accepts values that the user types at the keyboard.

The `cout` and `cin` operators offer the new C++ programmer input and output operators they can use with relative ease. Both of these operators have a limited scope, but they give you the ability to send output from and receive input to your programs. There are corresponding functions supplied with all C++ compilers called `printf()` and `scanf()`. These functions are still used by C++ programmers due to their widespread use in regular C programs.

This chapter introduces you to

- The `cout` operator
- Control operators
- The `cin` operator

- The `printf()` output function
- The `scanf()` input function

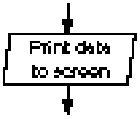
You will be surprised at how much more advanced your programs can be after you learn these input/output operators.

The **cout** Operator

`cout` sends output to the screen.

The `cout` operator sends data to the standard output device. The standard output device is usually the screen; you can, however, redirect standard output to another device. If you are unfamiliar with device redirection at the operating system level, don't worry, you learn more about it in this book. At this point, `cout` sends all output to the screen.

The format of the `cout` is different from those of other C++ commands. The format for `cout` is



```
cout << data [ << data ];
```

The `data` placeholder can be variables, literals, expressions, or a combination of all three.

Printing Strings

To print a string constant, simply type the string constant after the `cout` operator. For example, to print the string, `The rain in Spain`, you would simply type this:

Print the sentence `“The rain in Spain”` *to the screen.*

```
cout << “The rain in Spain”;
```

You must remember, however, that `cout` does not perform an automatic carriage return. This means the screen's cursor appears directly after the last printed character and subsequent `cout`s begin thereafter.

To better understand this concept, try to predict the output from the following three `cout` operators:

```
cout << “Line 1”;
cout << “Line 2”;
cout << “Line 3”;
```

These operators produce the following output:

```
Line 1Line 2Line 3
```

which is probably not what you intended. Therefore, you must include the newline character, `\n`, whenever you want to move the cursor to the next line. The following three `cout` operators produce a three-line output:

```
cout << “Line 1\n”;
cout << “Line 2\n”;
cout << “Line 3\n”;
```

The output from these `cout`s is

```
Line 1
Line 2
Line 3
```

The `\n` character sends the cursor to the next line no matter where you insert it. The following three `cout` operators also produce the correct three-line output:

```
cout << “Line 1”;
cout << “\nLine 2\n”;
cout “Line 3”;
```

The second `cout` prints a newline before it prints anything else. It then prints its string followed by another newline. The third string prints on the third line.

You also can print strings stored in character arrays by typing the array name inside the `cout`. If you were to store your name in an array defined as:

```
char my_name[ ] = “Lyndon Harris”;
```

you could print the name with the following `cout`:

```
cout << my_name;
```

The following section of code prints three string literals on three different lines:

```
cout << “Nancy Carson\n”;
cout << “1213 Oak Street\n”;
cout << “Fairbanks, Alaska\n”;
```

The `cout` is often used to label output. Before printing an age, amount, salary, or any other numeric data, you should print a string constant that tells the user what the number means. The following `cout` tells the user that the next number printed is an age. Without this `cout`, the user would not know what the number represented.

```
cout << “Here is the age that was found in our files:”;
```

You can print a blank line by printing two newline characters, `\n`, next to each other after your string, as in:

```
cout << “Prepare the invoices...\n\n”;
```

Examples

1. The following program stores a few values in three variables, then prints the results:

```
// Filename: C7PRNT1.CPP
// Prints values in variables.

#include <iostream.h>

main()
{
  char first = ‘E’;      // Store some character, integer,
  char middle = ‘W’;     // and floating-point variable.
  char last = ‘C’;
  int age = 32;
  int dependents = 2;
  float salary = 25000.00;
  float bonus = 575.25;

  // Prints the results.
  cout << first << middle << last;
  cout << age << dependents;
  cout << salary << bonus;
  return 0;
}
```

2. The last program does not help the user. The output is not labeled, and it prints on a single line. Here is the same program with a few messages included and some newline characters placed where needed:

```
// Filename: C7PRNT2.CPP
// Prints values in variables with appropriate labels.

#include <iostream.h>

main()
{
   char first = ‘E’;      // Store some character, integer,
   char middle = ‘W’;      // and floating-point variable.
   char last = ‘C’;
   int age = 32;
   int dependents = 2;
   float salary = 25000.00;
   float bonus = 575.25;

   // Prints the results.
   cout << “Here are the initials:\n”;
   cout << first << middle << last <<“\n”;
   cout << “The age and number of dependents are\n”;
   cout << age << “   “ << dependents << “\n\n”;
   cout << “The salary and bonus are\n”;
   cout << salary << ‘ ‘ << bonus;
   return 0;
}
```

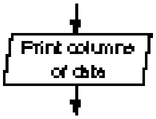
The output from this program appears below:

```
Here are the initials:
EWC
The age and number of dependents are
32   2
```

```
The salary and bonus are
25000 575.25
```

The first floating-point values do not print with zeros, but the number is correct. The next section shows you how to set the number of leading and trailing zeros.

3. If you have to print a table of numbers, you can use the `\t` tab character to do so. Place the tab character between each of the printed numbers. The following program prints a list of team names and number of hits for the first three weeks of the season:



```
// Filename: C7TEAM.CPP
// Prints a table of team names and hits for three weeks.

#include <iostream.h>

main()
{
  cout << “Parrots\tRams\tKings\tTitans\tChargers\n”;
  cout << “3\t5\t3\t1\t0\n”;
  cout << “2\t5\t1\t0\t1\n”;
  cout << “2\t6\t4\t3\t0\n”;
  return 0;
}
```

This program produces the table shown below. You can see that even though the names are different widths, the numbers print correctly beneath them. The `\t` character forces the next name or value to the next tab position (every eight characters).

```
Parrots Rams    Kings   Titans  Chargers
3       5       3       1       0
2       5       1       0       1
2       6       4       3       0
```

Control Operators

You have already seen the need for additional program-output control. All floating-point numbers print with too many decimal places for most applications. What if you want to print only dollars and cents (two decimal places), or print an average with a single decimal place?

You can modify the way numbers print.

You can specify how many print positions to use in printing a number. For example, the following `cout` prints the number 456, using three positions (the length of the data):

```
cout << 456;
```

If the 456 were stored in an integer variable, it would still use three positions to print because the number of digits printed is three. However, you can specify how many positions to print. The following `cout` prints the number 456 in five positions (with two leading spaces):

```
cout << setw(5) << setfill(‘ ‘) << 456;
```

You typically use the `setw` manipulator when you want to print data in uniform columns. Be sure to include the iomanip.h header file in any programs that use manipulators because iomanip.h describes how the `setw` works to the compiler.

The following program shows you the importance of the width number. Each `cout` output is described in the comment to its left.

```
// Filename: C7MOD1.CPP
// Illustrates various integer width cout modifiers.

#include <iostream.h>
#include <iomanip.h>

main()
{                               // The output appears below.
   cout << 456 << 456 << 456 << “\n”;   // Prints 456456456
   cout << setw(5) << 456 << setw(5) << 456 << setw(5) <<
        456 << “\n”;              //  Prints 456  456  456
   cout << setw(7) << 456 << setw(7) << 456 << setw(7) <<
        456 << “ \n”;       //  Prints 456    456    456
   return 0;
}
```

When you use a `setw` manipulator inside a conversion character, C++ right-justifies the number by the width you specify. When you specify an eight-digit width, C++ prints a value inside those eight digits, padding the number with leading blanks if the number does not fill the whole width.

NOTE: If you do not specify a width large enough to hold the number, C++ ignores your width request and prints the number in its entirety.

You can control the width of strings in the same manner with the `setw` manipulator. If you don't specify enough width to output the full string, C++ ignores the width. The mailing list application in the back of this book uses this technique to print names on mailing labels.

NOTE: `setw()` becomes more important when you print floating-point numbers.

`setprecision(2)` prints a floating-point number with two decimal places. If C++ has to round the fractional part, it does so. The following `cout`:

```
cout << setw(6) << setprecision(2) << 134.568767;
```

produces the following output:

```
134.57
```

Without the `setw` or `setprecision` manipulators, C++ would have printed:

```
134.568767
```

TIP: When printing floating-point numbers, C++ always prints the entire portion to the left of the decimal (to maintain as much accuracy as possible) no matter how many positions you specify. Therefore, many C++ programmers ignore the `setw` manipulator for floating-point numbers and only specify the precision, as in `setprecision(2)`.

Examples

1. If you want to control the width of your data, use a `setw` manipulator. The following program is a revision of the C7TEAM.CPP shown earlier. Instead of using the tab character, `\t`, which is limited to eight spaces, this program uses the width specifier to set the tabs. It ensures that each column is 10 characters wide.

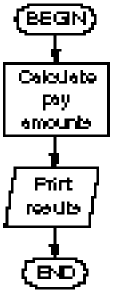
```
// Filename: C7TEAMMD.CPP
// Prints a table of team names and hits for three weeks
// using width-modifying conversion characters.

#include <iostream.h>
#include <iomanip.h>

main()
{
  cout << setw(10) << “Parrots” << setw(10) <<
       “Rams” << setw(10) << “Kings” << setw(10) <<
       “Titans” << setw(10) << “Chargers” << “\n”;
  cout << setw(10) << 3 << setw(10) << 5 <<
          setw(10) << 2 << setw(10) << 1 <<
          setw(10) << 0 << “\n”;
  cout << setw(10) << 2 << setw(10) << 5 <<
          setw(10) << 1 << setw(10) << 0 <<
          setw(10) << 1 << “\n”;
  cout << setw(10) << 2 << setw(10) << 6 <<
          setw(10) << 4 << setw(10) << 3 <<
          setw(10) << 0 << “\n”;
  return 0;
}
```

2. The following program is a payroll program. The output is in “dollars and cents” because the dollar amounts print properly to two decimal places.

```
// Filename: C7PAY1.CPP
// Computes and prints payroll data properly in dollars
// and cents.
```



```
#include <iostream.h>
#include <iomanip.h>

main()
{
  char emp_name[ ] = “Larry Payton”;
  char pay_date[ ] = “03/09/92”;
  int hours_worked = 43;
  float rate = 7.75;          // Pay per hour
  float tax_rate = .32;      // Tax percentage rate
  float gross_pay, taxes, net_pay;

  // Computes the pay amount.
  gross_pay = hours_worked * rate;
  taxes = tax_rate * gross_pay;
  net_pay = gross_pay - taxes;

  // Prints the results.
  cout << “As of: “ << pay_date << “\n”;
  cout << emp_name << “ worked “ << hours_worked <<
       “ hours\n”;
  cout << “and got paid “ << setw(2) << setprecision(2)
       << gross_pay << “\n”;
  cout << “After taxes of: “ << setw(6) << setprecision(2)
       << taxes << “\n”;
  cout << “his take-home pay was $” << setw(8) <<
           setprecision(2) << net_pay << “\n”;
  return 0;
}
```

The output from this program follows. Remember that the floating-point variables still hold the full precision (to six decimal places), as they did in the previous program. The modifying `setw` manipulators only affect how the variables are output, not what is stored in them.

```
As of: 03/09/92
Larry Payton worked 43 hours
and got paid 333.25
After taxes of: 106.64
his take-home pay was $226.61
```

3. Most C++ programmers do not use the `setw` manipulator when printing dollars and cents. Here is the payroll program again that uses the shortcut floating-point width method. Notice the previous three `cout` statements include no `setw` manipulator. C++ automatically prints the full number to the left of the decimal and prints only two places to the right.

```
// Filename: C7PAY2.CPP
// Computes and prints payroll data properly
// using the shortcut modifier.

#include <iostream.h>
#include <iomanip.h>

main()
{
  char emp_name[ ] = “Larry Payton”;
  char pay_date[ ] = “03/09/92”;
  int hours_worked = 43;
  float rate = 7.75;          // Pay per hour
  float tax_rate = .32;     // Tax percentage rate
  float gross_pay, taxes, net_pay;

  // Computes the pay amount.
  gross_pay = hours_worked * rate;
  taxes = tax_rate * gross_pay;
  net_pay = gross_pay - taxes;

  // Prints the results.
  cout << “As of: “ << pay_date << “\n”;
  cout << emp_name << “ worked “ << hours_worked <<
          “ hours\n”;
  cout << “and got paid “ << setprecision(2) << gross_pay
       << “\n”;
  cout << “After taxes of: “ << setprecision(2) << taxes
       << “\n”;
  cout << “his take-home pay was “ << setprecision(2) <<
          net_pay << “\n”;
  return 0;
}
```

This program's output is the same as the previous program's.

The `cin` Operator

You now understand how C++ represents data and variables, and you know how to print the data. There is one additional part of programming you have not seen: inputting data to your programs.

Until this point, you have not inputted data into a program. All data you worked with was assigned to variables in the program. However, this is not always the best way to transfer data to your programs; you rarely know what your data is when you write your programs. The data is known only when you run the programs (or another user runs them).

The `cin` operator stores keyboard input in variables.

The `cin` operator is one way to input from the keyboard. When your programs reach the line with a `cin`, the user can enter values directly into variables. Your program can then process those variables and produce output. Figure 7.1 illustrates the difference between `cout` and `cin`.

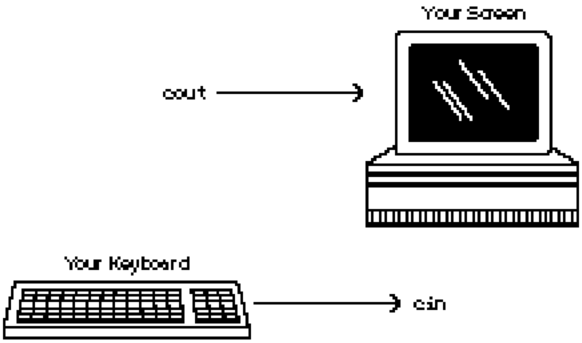


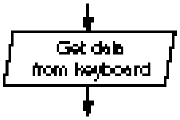
Figure 7.1. The actions of `cout` and `cin`.

The `cin` Function Fills Variables with Values

There is a major difference between `cin` and the assignment statements (such as `i = 17;`). Both fill variables with values. However, the assignment statement assigned specific values to variables at programming time. When you run a program with assignment statements, you know from the program's listing exactly what values go into the variables because you wrote the program specifically to store those values. Every time you run the program, the results are exactly the same because the same values are *assigned* to the same variables.

You have no idea, when you write programs that use `cin`, what values will be assigned to the `cin`'s variables because their values are not known until the program runs and the user enters those values. This means you have a more flexible program that can be used by a variety of people. Every time the program is run, different results are created, depending on the values typed at each `cin` in the program.

The `cin` has its drawbacks. Therefore, in the next few chapters you will use `cin` until you learn more powerful (and flexible) input methods. The `cin` operator looks much like `cout`. It contains one or more variables that appear to the right of the operator name. The format of the `cin` is



```
cin >> value [>> values];
```

The iostream.h header file contains the information C++ needs to use `cin`, so include it when using `cin`.

NOTE: The `cin` operator uses the same manipulators (`setw` and `setprecision`) as the `cout` operator.

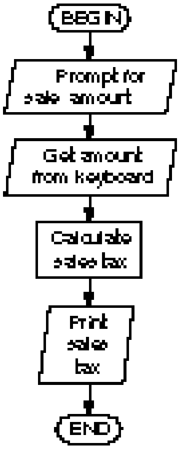
As mentioned earlier, `cin` poses a few problems. The `cin` operator requires that your user type the input *exactly* as `cin` expects it. Because you cannot control the user's typing, this cannot be ensured. You might want the user to enter an integer value followed by a floating-point value and your `cin` operator call might expect it too, but your user might decide to enter something else! If this happens, there is not much you can do because the resulting input is incorrect and your C++ program has no reliable method for testing user accuracy. Before every `cin`, print a prompt that explains exactly what you expect the user to type.

For the next few chapters, you can assume that the user knows to enter the proper values, but for your "real" programs, read on for better methods to receive input, starting with Chapter 21, "Device and Character Input / Output."

The `cin` operator requires that the user type correct input. This is not always possible to guarantee!

Examples

1. If you wanted a program that computed a seven percent sales tax, you could use the `cin` statement to figure the sales, compute the tax, and print the results as the following program shows:



```
// Filename: C7SLTX1.CPP
// Prompt for a sales amount and print the sales tax.

#include <iostream.h>
#include <iomanip.h>

main()
{
   float total_sale;     // User's sale amount goes here.
   float stax;

   // Display a message for the user.
   cout << "What is the total amount of the sale? ";

   // Receive the sales amount from user.
   cin >> total_sale;

   // Calculate sales tax.
   stax = total_sale * .07;
```

```
  cout << “The sales tax for “ << setprecision(2) <<
          total_sale << “ is “ << setprecision (2) << stax;
  return 0;
}
```

Because the first `cout` does not contain a newline character, `\n`, the user’s response to the prompt appears to the right of the question mark.

2. When inputting keyboard strings into character arrays with `cin`, you are limited to receiving one word at a time. The `cin` does not enable you to type more than one word in a single character array at a time. The following program asks the user for his or her first and last name. The program has to store those two names in two different character arrays because `cin` cannot input both names at once. The program then prints the names in reverse order.

```
// Filename: C7PHON1.CPP
// Program that requests the user’s name and prints it
// to the screen as it would appear in a phone book.

#include <iostream.h>
#include <iomanip.h>

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

  cout << “What is your first name? “;
  cin >> first;
  cout << “What is your last name? “;
  cin >> last;
  cout << “\n\n”;      // Prints two blank lines.
  cout << “In a phone book, your name would look like this:\n”;
  cout << last << “, “ << first;
  return 0;
}
```

3. Suppose you want to write a program that does simple addition for your seven-year-old daughter. The following program prompts her for two numbers. The program then waits for her to type an answer. When she gives her answer, the program displays the correct result so she can see how well she did.

```
// Filename: C7MATH.CPP
// Program to help children with simple addition.
// Prompt child for two values after printing
// a title message.
#include <iostream.h>
#include <iomanip.h>

main()
{
  int num1, num2, ans;
  int her_ans;

  cout << "*** Math Practice ***\n\n\n";
  cout << "What is the first number? ";
  cin >> num1;
  cout << "What is the second number? ";
  cin >> num2;

  // Compute answer and give her a chance to wait for it.
  ans = num1 + num2;

  cout << "\nWhat do you think is the answer? ";
  cin >> her_ans;      // Nothing is done with this.

  // Prints answer after a blank line.
  cout << "\n" << num1 << " plus " << num2 << " is "
       << ans << "\n\nHope you got it right!";
  return 0;
}
```

printf() and scanf()

Before C++, C programmers had to rely on function calls to perform input and output. Two of those functions, `printf()` and `scanf()`, are still used frequently in C++ programs, although `cout` and `cin` have advantages over them. `printf()` (like `cout`) prints values to the screen and `scanf()` (like `cin`) inputs values from the keyboard. `printf()` requires a controlling format string that describes the data you want to print. Likewise, `scanf()` requires a controlling format string that describes the data the program wants to receive from the keyboard.

> **NOTE:** `cout` is the C++ replacement to `printf()` and `cin` is the C++ replacement to `scanf()`.

Because you are concentrating on C++, this chapter only briefly covers `printf()` and `scanf()`. Throughout this book, a handful of programs use these functions to keep you familiar with their format. `printf()` and `scanf()` are not obsolete in C++, but their use will diminish dramatically when programmers move away from C and to C++. `cout` and `cin` do not require controlling strings that describe their data; `cout` and `cin` are intelligent enough to know how to treat data. Both `printf()` and `scanf()` are limited—especially `scanf()`—but they do enable your programs to send output and to receive input.

The printf() Function

The `printf()` function sends output to the screen.

`printf()` sends data to the standard output device, which is generally the screen. The format of `printf()` is different from those of regular C++ commands. The values that go inside the parentheses vary, depending on the data you are printing. However, as a general rule, the following `printf()` format holds true:

```
printf(control_string [, one or more values]);
```

Notice `printf()` always requires a `control_string`. This is a string, or a character array containing a string, that determines how the rest of the values (if any are listed) print. These values can be variables, literals, expressions, or a combination of all three.

TIP: Despite its name, `printf()` sends output to the screen and not to the printer.

The easiest data to print with `printf()` are strings. To print a string constant, you simply type that string constant inside the `printf()` function. For example, to print the string `The rain in Spain`, you would simply type the following:

Print the phrase `“The rain in Spain”` *to the screen.*

```
printf(“The rain in Spain”);
```

`printf()`, like `cout`, does *not* perform an automatic carriage return. Subsequent `printf()`s begin next to that last printed character. If you want a carriage return, you must supply a newline character, as so:

```
printf(“The rain in Spain\n”);
```

You can print strings stored in character arrays also by typing the array name inside the `printf()`. For example, if you were to store your name in an array defined as:

```
char my_name[] = “Lyndon Harris”;
```

you could print the name with this `printf()`:

```
printf(my_name);
```

You must include the stdio.h header file when using `printf()` and `scanf()` because stdio.h determines how the input and output functions work in the compiler. The following program assigns a message in a character array, then prints that message.

```
// Filename: C7PS2.CPP
// Prints a string stored in a character array.
#include <stdio.h>
main()
{
   char message[] = “Please turn on your printer”;
   printf(message);
   return 0;
}
```

Conversion Characters

Inside most `printf()` control strings are *conversion characters.* These special characters tell `printf()` exactly how the data (following the characters) are to be interpreted. Table 7.1 shows a list of common conversion characters. Because any type of data can go inside the `printf()`'s parentheses, these conversion characters are required any time you print more than a single string constant. If you don't want to print a string, the string constant must contain at least one of the conversion characters.

Table 7.1. Common **`printf()`** conversion characters.

Conversion Character	*Output*
`%s`	String of characters (until null zero is reached)
`%c`	Character
`%d`	Decimal integer
`%f`	Floating-point numbers
`%u`	Unsigned integer
`%x`	Hexadecimal integer
`%%`	Prints a percent sign (%)

Note: You can insert an `l` *(lowercase* `l`*) or* `L` *before the integer and floating-point conversion characters (such as* `%ld` *and* `%Lf`*) to indicate that a long integer or long double floating-point is to be printed.*

NOTE: Characters other than those shown in the table print exactly as they appear in the control string.

When you want to print a numeric constant or variable, you must include the proper conversion character inside the `printf()` control string. If `i`, `j`, and `k` are integer variables, you cannot print them with the `printf()` that follows.

```
printf(i,j,k);
```

Because `printf()` is a function and not a command, this `printf()` function has no way of knowing what type the variables are. The results are unpredictable, and you might see garbage on your screen—if anything appears at all.

When you print numbers, you must first print a control string that includes the format of those numbers. The following `printf()` prints a string. In the output from this line, a string appears with an integer (`%d`) and a floating-point number (`%f`) printed inside that string.

```
printf("I am Betty, I am %d years old, and I make %f\n",
        35, 34050.25);
```

This produces the following output:

```
I am Betty, I am 35 years old, and I make 34050.25
```

Figure 7.2 shows how C interprets the control string and the variables that follow. Be sure you understand this example before moving on. It is the foundation of the `printf()` function.

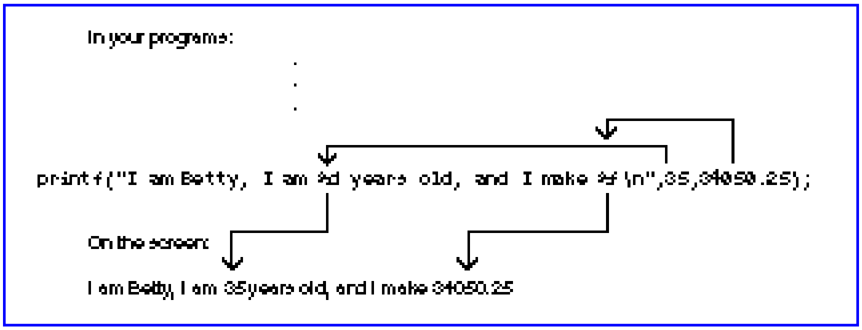

In your programs:
.
.
.
printf("I am Betty, I am %d years old, and I make %f\n",35,34050.25);
On the screen:
I am Betty, I am 35 years old, and I make 34050.25

Figure 7.2. Control string in action.

You also can print integer and floating-point variables in the same manner.

Examples

1. The following program stores a few values in three variables, then prints the results.

```
// Filename: C7PRNTF.CPP
// Prints values in variables with appropriate labels.
#include <stdio.h>

main()
{
   char first=’E’;         // Store some character, integer,
   char middle=’W’;        // and floating-point variable.
   char last=’C’;
   int age=32;
   int dependents=2;
   float salary=25000.00;
   float bonus=575.25;

   /* Prints the results. */
   printf(“Here are the initials\n”);
   printf(“%c%c%c\n\n”, first, middle, last);
   printf(“The age and number of dependents are\n”);
   printf(“%d   %d\n\n”, age, dependents);
   printf(“The salary and bonus are\n”);
   printf(“%f %f”, salary, bonus);
   return 0;
}
```

The output from this program is

```
Here are the initials
EWC

The age and number of dependents are
32   2

The salary and bonus are
25000.000000 575.250000
```

2. The floating-point values print with too many zeros, of course, but the numbers are correct. You can limit the number of leading and trailing zeros that is printed by adding a width specifier in the control string. For instance, the following `printf()` prints the `salary` and `bonus` with two decimal places:

```
printf(“%.2f %.2f”, salary, bonus);
```

Make sure your printed values match the control string supplied with them. The `printf()` function cannot fix problems resulting from mismatched values and control strings. Don't try to print floating-point values with character-string control codes. If you list five integer variables in a `printf()`, be sure to include five `%d` conversion characters in the `printf()` as well.

Printing ASCII Values

There is one exception to the rule of printing with matching conversion characters. If you want to print the ASCII value of a character, you can print that character (whether it is a constant or a variable) with the integer `%d` conversion character. Instead of printing the character, `printf()` prints the matching ASCII number for that character.

Conversely, if you print an integer with a `%c` conversion character, you see the character that matches that integer's value from the ASCII table.

The following `printf()`s illustrate this fact:

```
printf("%c", 65);  // Prints the letter A.
printf("%d", 'A'); // Prints the number 65.
```

The scanf() Function

The `scanf()` function stores keyboard input to variables.

The `scanf()` function reads input from the keyboard. When your programs reach the line with a `scanf()`, the user can enter values directly into variables. Your program can then process the variables and produce output.

The `scanf()` function looks much like `printf()`. It contains a control string and one or more variables to the right of the control string. The control string informs C++ exactly what the incoming keyboard values look like, and what their types are. The format of `scanf()` is

```
scanf(control_string, one or more values);
```

The `scanf()` `control_string` uses almost the same conversion characters as the `printf()` `control_string`, with two slight differences. You should never include the newline character, `\n`, in a `scanf()` control string. The `scanf()` function “knows” when the input is finished when the user presses Enter. If you supply an additional newline code, `scanf()` might not terminate properly. Also, always put a beginning space inside every `scanf()` control string. This does not affect the user’s input, but `scanf()` sometimes requires it to work properly. Later examples in this chapter clarify this fact.

The `scanf()` function requires that your user type accurately. This is not always possible to guarantee!

As mentioned earlier, `scanf()` poses a few problems. The `scanf()` function requires that your user type the input exactly the way `control_string` specifies. Because you cannot control your user’s typing, this cannot always be ensured. For example, you might want the user to enter an integer value followed by a floating-point value (your `scanf()` control string might expect it too), but your user might decide to enter something else! If this happens, there is not much you can do. The resulting input is incorrect, but your C program has no reliable method for testing user accuracy before your program is run.

> **CAUTION:** The user’s keyboard input values *must* match, in number and type, the control string contained in each `scanf()`.

Another problem with `scanf()` is not as easy for beginners to understand as the last. The `scanf()` function requires that you use pointer variables, not regular variables, in its parentheses. Although this sounds complicated, it doesn’t have to be. You should have no problem with `scanf()`’s pointer requirements if you remember these two simple rules:

1. Always put an ampersand (`&`) before variable names inside a `scanf()`.
2. Never put an ampersand (`&`) before an array name inside a `scanf()`.

Despite these strange `scanf()` rules, you can learn this function quickly by looking at a few examples.

Examples

1. If you want a program that computes a seven percent sales tax, you could use the `scanf()` statement to receive the sales, compute the tax, and print the results as the following program shows.

```
// Filename: C7SLTXS.CPP
// Compute a sales amount and print the sales tax.
#include <stdio.h>
main()
{
   float total_sale;   // User's sale amount goes here.
   float stax;

   // Display a message for the user.
   printf(“What is the total amount of the sale? “);

   // Compute the sales amount from user.
   scanf(“ %f”, &total_sale);   // Don't forget the beginning
                                // space and an &.

   stax = total_sale * .07;  // Calculate the sales tax.

   printf(“The sales tax for %.2f is %.2f”, total_sale, stax);
   return 0;
}
```

If you run this program, the program waits for you to enter a value for the total sale. Remember to use the ampersand in front of the `total_sale` variable when you enter it in the `scanf()` function. After pressing the Enter key, the program calculates the sales tax and prints the results.

If you entered 10.00 as the sale amount, you would receive the following output :

```
The sales tax for 10.00 is 0.70
```

2. Use the string `%s` conversion character to input keyboard strings into character arrays with `scanf()`. As with `cin`, you are limited to inputting one word at a time, because you

cannot type more than one word into a single character array with `scanf()`. The following program is similar to C7PHON1.CPP except the `scanf()` function, rather than `cin`, is used. It must store two names in two different character arrays, because `scanf()` cannot input both names at once. The program then prints the names in reverse order.

```
// Filename: C7PHON2.CPP
// Program that requests the user’s name and prints it
// to the screen as it would appear in a phone book.
#include <stdio.h>
main()
{
   char first[20], last[20];
   printf(“What is your first name? “);
   scanf(“ %s”, first);
   printf(“What is your last name? “);
   scanf(“ %s”, last);
   printf(“\n\n”);    // Prints two blank lines.
   printf(“In a phone book, your name would look like”
          “this:\n”);
   printf(“%s, %s”, last, first);
   return 0;
}
```

3. How many values are entered with the following `scanf()`, and what are their types?

```
scanf(“ %d %d %f %s”, &i, &j, &k, l);
```

Review Questions

The answers to the Review Questions are in Appendix B.

1. What is the difference between `cout` and `cin`?
2. Why is a prompt message important before using `cin` for input?

3. How many values do you enter with the following `cin`?

```
cin >> i >> j >> k >> l;
```

4. Because they both assign values to variables, is there any difference between assigning values to variables and using `cin` to give them values?

5. True or false: The `%s` conversion character is usually not required in `printf()` control strings.

6. Which types of variables do not require the ampersand (`&`) character in `scanf()` functions?

7. What is the output produced by the following `cout`?

```
cout << “The backslash \”\\\” character is special”;
```

8. What is the result of the following `cout`?

```
cout << setw(8) << setprecision(3) << 123.456789;
```

Review Exercises

1. Write a program that prompts the user for his or her name and weight. Store these values in separate variables and print them on-screen.

2. Assume you are a college professor and have to average grades for 10 students. Write a program that prompts you for 10 different grades, then displays an average of them.

3. Modify the program in Exercise 2 to ask for each student’s name as well as her grade. Print the grade list to the screen, with each student’s name and grade in two columns. Make sure the columns align by using a `setw` manipulator on the grade. At the bottom, print the average of the grades. (*Hint:* Store the 10 names and 10 grades in different variables with different names.) This program is easy, but takes thirty or so lines, plus appropriate comments and prompts. Later, you learn ways to streamline this program.

4. This exercise tests your understanding of the backslash conversion character: Write a program that uses `cout` operators to produce the following picture on-screen:

```
                                 +
                                /*\
                                |||
             *                  |||
            **                  |||
       /\ __*                   |||
      /  \||                   /|||\
    /      |                  /  *  \
  / |======|\                   ***
    | +  + |                     *
    |  ||  |
____|_+||+_|______________/================\_______
```

Summary

You now can print almost anything to the screen. By studying the manipulators and how they behave, you can control your output more thoroughly than ever before. Also, because you can receive keyboard values, your programs are much more powerful. No longer do you have to know your data values when you write the program. You can ask the user to enter values into variables with `cin`.

You have the tools to begin writing programs that fit the data processing model of INPUT->PROCESS->OUTPUT. This chapter concludes the preliminary discussion of the C++ language. This part of the book attempted to give you an overview of the language and to teach you enough of the language elements so you can begin writing helpful programs.

Chapter 8, “Using C++ Math Operators and Precedence,” begins a new type of discussion. You learn how C++’s math and relational operators work on data, and the importance of the precedence table of operators.