



Module 814

Input and Output in C

Aim

After working through this module you should be able to create C programs that fully utilise the input/output libraries to read data from the keyboard and to display data on the computer monitor.

Learning objectives

After working through this module you should be able to:

1. Identify and describe the purpose of the C standard input/output library.
2. Use the standard input operations in C programs.
3. Use the standard output operations in C programs.
4. Convert data values to formatted text using standard output operations.
5. State the significance of standard error output.
6. Use other input/output libraries, for example the Turbo C console library, to perform input and output operations.

Content

The standard input/output library.

Standard input operations.

Standard output operations.

Standard error output.

Other input/output libraries.

Screen output using CONIO.

Learning Strategy

Read the printed module and the assigned readings and complete the exercises as requested.

Assessment

Completion of exercises and the CML test at the end of the module.

References & resources

The C Programming Language. 2nd. edition
Brian W. Kernighan and Dennis M. Ritchie
Prentice-Hall, 1988

Turbo C/C++ Manuals.

Turbo C/C++ MS DOS compiler.

Introduction to Module 814

In this module you will be introduced to the methods and practices used by C programmers to perform input and output. In the previous module (Module 813: Fundamental Programming Structures in C) you discovered the fundamental elements of a C program. This module describes in more detail the standard input/output library that is common to all C compilers.

C is not a language rich in input and output facilities and the functions that are provided are very primitive. Programmers therefore rely extensively on program, and in particular input/output, libraries for support in developing the user interface for a program. The standard input/output library is, by its very nature, standard and must therefore be suited to any hardware platform. Consequently the functions that it provides do not support colour monitors because you cannot guarantee that the hardware platform has a colour monitor. Similarly, the functions do not support mouse input, text-based or high resolution graphics output, or other “non-standard” input and output devices.

For this reason many programmers create, or purchase from third-party vendors, input/output libraries that are specific to the hardware platform and interface tasks required by the program. Many libraries exist to perform functions such as the creation and management of menus, high-resolution graphics, windows, forms and so on. One library that will be looked at in this module is the console input/output (CONIO) library that is provided with the Turbo C development system. This library extends the function provided in STDIO to include screen management operations for text-based output.

These notes make use of a number of program examples. Each example is provided on a floppy disk which is part of the unit materials. You should load, compile and run these programs to discover how they work. You can also use these examples as prototypes for the design of your own programs. In order to conserve paper, the programs have been written with a minimum of documentation. Your programs would normally be developed with more substantial documentation.

Objective 1 After working through this module you should be able to identify and describe the purpose of the C standard input/output library.

STANDARD INPUT AND OUTPUT

Each hardware platform interacts with the environment in its own way. Different peripheral devices have different modes of communication. The operating system may, or may not, perform a variety of input and output functions. The designers of C were faced with the dilemma of including input and output facilities, and therefore writing a separate compiler for each hardware platform, or creating a universal language that does not include the hardware specific input and output operations. The designers chose the latter approach and opted to define, but not implement, a standard set of input and output operations. These operations form part of the standard C library. Any C program, as long as it uses these standard operations, can be compiled and run on any hardware platform as long as the library routines are available on that platform.

The header file `STDIO.H` and its associated object code file define the standard input and output operations. This file forms part of the larger C standard library defined in the ANSI standard for the language. The standard library consists of 12 library files, each designated to a particular purpose. The standard library includes data structures and functions to:

- handle strings, as defined in `STRING.H`
- perform character tests and conversions, as defined in `CTYPE.H`
- perform mathematical operations, as defined in `MATH.H`
- perform number conversions, storage allocation and similar tasks, as defined in `STDLIB.H`
- manipulate date and time data, as defined in `TIME.H`

The standard input and output library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the hardware and operating system platform does not operate in this way, the library does whatever necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output. Standard input and output are special cases of text file input and output. As such, many of the functions used for standard input and output have equivalent functions that are used with files. This will be explored further in Module 816: File Access in C.

Module 814 Input and Output in C

In many environments, a file may be substituted for the keyboard by using the `<` convention for input redirection. For example, the command line:

```
prog < infile
```

will run the executable file called `prog` with all input coming from the text file called `infile`. Similarly, a file may be substituted for monitor output by using the `>` convention for output redirection. For example, the command line:

```
prog > outfile
```

will run the executable file called `prog` with all output going to the text file called `outfile`. It is possible to redirect both input and output simultaneously, as shown by the example:

```
prog < infile > outfile
```

The facility for redirection comes from two components. The C language has defined input and output to be stream based and therefore the substitution of one file input with another is transparent to the library system. The second component is the operating system itself. Most operating systems also define input and output as text streams and therefore substituting files for keyboards, or files for monitors, is a possible and desirable characteristic.

One consequence of the text stream input and output model is that the facilities provided are plain, and fundamentally provide only terminal-like facilities. The use of characteristics such as colour, screen positioning, mouse or scanner input, graphics, and sound are not part of this stream based model. These facilities are not found in the standard library. If the user interface requires these features then another library – either user defined or third party – must be used.

Spend some time studying the file named `STDIO.H`. There will be a lot in it that is difficult to understand, but parts of it will look familiar. The name `STDIO.H` is a cryptic form of the name standard input/output header, because that is exactly what it does. It defines the standard input and output functions in the form of `#defines` and macros. Don't worry too much about the details of this now – you can always return to this topic later for more study if it interests you – but you will really have no need to completely understand the `STDIO.H` file. You will have a tremendous need to use it however, so these comments on its use and purpose are necessary.

The first program in this section [`SIMPLEIO.C`] is also the first formal study of a program with *standard I/O*. Standard I/O refers to the most usual places where data is either read from, the keyboard, or written to,

Module 814 Input and Output in C

the video monitor. Since they are used so much, they are used as the default I/O devices and do not need to be named in the Input/Output instructions. This will make more sense when we actually start to use them so let's look at the program.

```
#include "stdio.h"           /* standard header for input/output */
void main( )
{
    char c;
    printf("Enter any characters, X = halt program.\n");
    do {
        c = getchar( );      /* get a single character from the kb */
        putchar(c);         /* display the character on the monitor */
    } while (c != 'X');     /* until an X is hit */
    printf("\nEnd of program.\n");
}
```

The first line of the program uses an *#include* statement. This is very much like the *#define* studied in the previous modeule, except that instead of a simple substitution, an entire file is read in at this point. The system will find the file named **stdio.h** and read its entire contents in, replacing this statement. The file named **stdio.h** must contain valid C source statements that can be compiled as part of a program. This particular file is composed of several standard *#defines* to define some of the standard I/O operations. The file is called a *header file* and you will find several different header files on the source disks that came with your C compiler. Each of the header files has a specific purpose and any or all of them can be included in any program.

A header file contains only the definitions and prototypes of data structures and functions provided by the library. The source code for the library will be contained elsewhere or, in the case of third-party libraries, may not be provided at all. The library **must**, however, be available as an *object file* that will be linked with the program during the linking phase of the compilation process. A program may include many header files thus referencing a large number of program libraries.

The C compiler uses the double quote marks to indicate that the search for the include file will begin in the current directory, and if it is not found there, the search will continue in the directory specified for program libraries set up in the Turbo C environment.

It is also possible to use the less than (<) and greater than (>) symbols to indicate that the search for the header file should begin in the directory specified in the environment. In these circumstances the current directory is not searched at all thereby increasing the compilation speed slightly. For standard libraries it is possible to use either form of notation since the standard libraries are not normally found in the current directory. For user-defined libraries it is preferable to use the quotation symbol notation since the libraries may reside with the program.

Most of the programs in this unit use the quotation symbol notation in the include statements. The next program uses < and > to illustrate the usage. Note that this will result in a slightly faster (but probably unnoticeable) compilation because the system will not bother to search the current directory.

Input/Output programming in C

C was not designed to be used as a language for lots of input and output, but as a systems language where a lot of internal operations are required. You would do well to use another language for I/O intensive programming, but C could be used if you desire. The keyboard input is very flexible, allowing you to get at the data in a very low level way, but very little help is given you. It is therefore up to you to take care of all of the bookkeeping chores associated with your required I/O operations. This may seem like a real pain in the neck, but in any given program, you only need to define your input routines once and then use them as needed. Don't let this worry you; as you gain experience with C, you will easily handle your I/O requirements.

Exercise 1

Modify the SIMPLEIO.C program to terminate input when the user enters a '\$' symbol.

Objective 2 After working through this module you should be able to use the standard input operations in C programs.

STANDARD INPUT OPERATIONS

Character Input

The simplest input mechanism is to read one character at a time from the *standard input*, normally the keyboard, with `getchar`.

```
int getchar(void)
```

Returning to the `SIMPLEIO.C` program introduced in the previous section. Just one variable – *c* – is defined and a message is printed out with the familiar `printf` function. A continuous loop follows which continues as long as *c* is not equal to (capital) X. If there is any question in your mind about the loop control, you should review Module 813 (Fundamental Programming Structures in C) before continuing. The two new functions within the loop are of paramount interest in this program since they are the new functions. These are functions to read a character from the keyboard and display it on the monitor one character at a time.

The function `getchar()` reads a single character from the standard input device, the keyboard, and assigns it to the variable *c*. The next function `putchar(c)`, uses the standard output device, the video monitor, and outputs the character contained in the variable *c*. The character is output at the current cursor location and the cursor is advanced one space for the next character. The loop continues reading and displaying characters until a capital X is entered which terminates the loop.

Compile and run this program for a few surprises. When you type on the keyboard, you will notice that what you type is displayed faithfully on the screen. When you hit the Enter key, the entire line is repeated. In fact, should only output each character once but it seems to be saving the characters up and redisplaying the line; a short explanation is required.

DOS is helping us (or getting in the way)

A little bit about how DOS works is needed to understand what is happening here. When a keypress is read from the keyboard, under DOS control, the characters are stored in a *buffer*. When the Enter key is detected, the entire string of characters is given to the program. When the characters are being typed, however, DOS will display the characters on the monitor. This is called *echo*, and happens in many of the applications you run.

With the above paragraph in mind, it should be clear that when you are typing a line of data into the program, the characters are being echoed by DOS, and when you press the Enter key, the characters are given to the program one at a time. As each character is given to the program, it displays it on the screen resulting in a repeat of the line typed in. To better illustrate this, type a line with a capital X somewhere in the middle of the line. You can type as many characters as you like following the X and they will all be displayed because the characters are being read in, echoed to the monitor, and placed in the DOS input buffer by the operating system.

To DOS of course, there is nothing special about a capital X. When the string is given to the program, however, the characters are accepted by the program one at a time and sent to the monitor one at a time, until a capital X is encountered. After the capital X is displayed, the loop is terminated, and the program is terminated. The characters on the input line following the capital X are not displayed because the capital X signalled program termination.

Compile and run the program making sure that you test the program completely. Don't get discouraged by this seemingly weird behaviour of the I/O system. These functions form the base level of input and output operations provided by STDIO. Other input and output functions are available but sometimes you will need to build your own high-level functions from these primitive operations.

A strange I/O method

Let us turn to another method of character I/O. The [SINGLEIO.C] program performs exactly the same purpose as the last and so the structure of the two programs are almost identical. Once again, we start with the standard I/O header file, this time using the < > method of locating it. A character variable named *c* is defined, and finally a welcoming message is printed. Like the last program, a loop is initiated that will continue to execute until a capital X is entered. The output of this program, however, is a little different to the previous example.

```
#include <stdio.h>
void main( )
{
char c;
printf("Enter any characters, terminate program with X\n");
do {
c = getch( );          /* get a character      */
putchar(c);           /* display the hit key  */
} while (c != 'X');
printf("\nEnd of program.\n");
}
```

The `getch()` is a new function that is also a get character function. It differs from `getchar()` in that it does not get tied up in DOS. It reads the character in without echo, and puts it directly into the program

Module 814 Input and Output in C

where it is operated on immediately. This function therefore reads a character, immediately displays it on the screen, and continues the operation until a capital X is typed.

When you compile and run this program, you will find that there is no repeat of the lines when you press the Enter key, and when you hit the capital X, the program terminates immediately. No Enter key is needed to get the program to accept the line with the X in it. If you look closely at the output, however, you will notice another problem. No linefeed is sent to the monitor when the Enter key is pressed and so no new line is started.

It is not apparent when using in most application programs, but when the Enter key is pressed, the program supplies a linefeed to go with the carriage return. A carriage return means that the output needs to return to the left side of the monitor and at the same time, it also needs to drop down a line. The linefeed is not automatic. The example [BETTERIN.C] improves the program by accomplishing this task.

```
#include "stdio.h"
#define CR 13                /* this defines CR to be 13 */
#define LF 10               /* this defines LF to be 10 */
void main( )
{
    char c;
    printf("Input any characters, hit X to stop.\n");
    do {
        c = getch( );        /* get a character */
        putchar(c);         /* display the hit key */
        if (c == CR) putchar(LF);
                            /* if it is a carriage return put out a linefeed too*/
    } while (c != 'X');
    printf("\nEnd of program.\n");
}
```

Two additional statements have been placed at the beginning of the program that define the character codes for the linefeed (LF), and the carriage return (CR). In the main program, after outputting the character, it is compared to CR, and if it is equal to CR, a linefeed character is also sent to the output. Note that the two #define statements can be omitted from the program and the character codes used directly, for example,

```
if (c == 13) putchar(10);
```

This technique, however is not very informative and makes it difficult for someone else to read. The method used in the program represents better programming practice.

Compiling and running this program should display exactly what you type in, including a linefeed with each carriage return, and should stop immediately when you type a capital X.

Which method?

Two methods of reading characters into a C program have been examined. Programmers are faced with a choice of which one they should use. It really depends on the application because each method has advantages and disadvantages.

When using the first method, the operating system is actually doing all of the work by storing the characters in an input buffer and signalling when a full line has been entered. The advantage of allowing DOS to perform the actual input is that the computer could be performing other tasks whilst input is occurring. However, single keystrokes cannot be entered with this technique because DOS would not report a buffer of characters to us until it recognised a carriage return.

The second method allows programs to get a single character, and act on it immediately. The program does not have to wait until DOS delivers a line of characters. The program must wait for the input keystroke thus requiring a loop. The computer cannot perform any other tasks while it is waiting for input. This method is useful for highly interactive types of program interfaces.

It should be mentioned at this point that there is also an *ungetch* function that works with the *getch* function. If a character is obtained using the *getch*() function and it is found that too many characters have been read, then *ungetch*() can be used to place the character back on the input device. This simplifies some programs because you often don't know that you don't want the character until you get it. You can only *ungetch* one character back to the input device, but that is sufficient to accomplish the task this function was designed for. It is difficult to demonstrate this function in a simple program so its use will be up to you to study when you need it.

The discussion so far in this section should be a good indication that, while the C programming language is very flexible, it does put a lot of responsibility on you as the programmer to keep many details in mind.

Reading integers

Most input required by programs is not single character input. Usually programs require numeric or string based input. This type of input is called *formatted input* and the C library provides a mechanism to achieve this kind of input.

The program [INTIN.C] is an example of how to read in some formatted data. The structure of this program is very similar to the last three except that it defines an int type variable and the loop continues until the variable somehow acquires a value of 100.

Module 814 Input and Output in C

```
#include "stdio.h"
void main( )
{
int valin;
printf("Input a number from 0 to 32767, stop with 100.\n");
do {
scanf("%d",&valin);          /* read a single integer value in */
printf("The value is %d\n", valin);
} while (valin != 100);
printf("End of program\n");
}
```

Instead of reading in a character at a time, as has occurred in the last three programs, an entire integer value is read with one call using the function named *scanf*. This function is very similar to the *printf* that has been used previously, except that it is used for input instead of output.

Examine the line with the *scanf* and you will notice that it does not ask for the variable *valin* directly, but gives the address of the variable since it expects to have a value returned from the function. Remember that a function must have the address of a variable in order to return a value to that variable in the calling program. Failing to supply a pointer in the *scanf* function is probably the most common problem encountered in using this function.

The function *scanf* reads the input line until it finds the first data field. It ignores leading blanks, and in this case it reads integer characters until it finds a blank or an invalid decimal character, at which time it stops reading and returns the value.

Remembering the discussion about the way the DOS input buffer works, it should be clear that nothing is actually acted on until a complete line is entered and it is terminated by a carriage return. At this time, the buffer becomes the input line, and the program will search the line reading all integer values it can find until the line is completely scanned. This is because the program is in a loop and the program is instructed to find a value, print it, find another, print it, etc. If you enter several values on one line, it will read each one in succession and display the values. Entering the value of 100 will cause the program to terminate, and entering the value 100 with other values following, will cause termination before the following values are considered.

Occasional errors

If you enter a number up to and including 32767 it will display correctly, but if you enter a larger number, it will appear to make an error. For example, if you enter the value 32768, it will display the value of -32768, entering the value 65536 will display as a zero. These are not errors but are caused by the way an integer is defined. The most significant bit of the 16 bit pattern available for the integer variable is the sign bit, so there are only 15 bits left for the value. The variable can

therefore only have the values from -32768 to 32767, any other values are outside the range of integer variables. This is for you to take care of in your programs. It is another example of the increased responsibility you must assume using C.

Compile and run this program, entering several numbers on a line to see the results, and with varying numbers of blanks between the numbers. Try entering numbers that are too big to see what happens, and finally enter some invalid characters to see what the system does with nondecimal characters.

Character string input

The next program [STRINGIN.C] is an example of reading a string variable. This program is identical to the last one except that instead of an integer variable, we have defined a string variable with an upper limit of 24 characters (remember that a string variable must have a null character at the end).

```
#include "stdio.h"
void main( )
{
    char big[25];
    printf("Input a character string, up to 25 characters.\n");
    printf("An X in column 1 causes the program to stop.\n");
    do {
        scanf("%s",big);
        printf("The string is -> %s\n", big);
    } while (big[0] != 'X');
    printf("End of program.\n");
}
```

The variable in the scanf does not need an & (pointer declaration) because big is an array variable and **by definition** it is already a pointer. This program should require no additional explanation; compile and run it to see if it works the way you expect.

You probably got a surprise when you ran it because it separated your sentence into separate words. When used in the string mode of input, scanf reads characters into the string until it comes to either the end of a line or a blank character. Therefore, it reads a word, finds the blank following it, and displays the result. Since we are in a loop, this program continues to read words until it exhausts the DOS input buffer. We have written this program to stop whenever it finds a capital X in column 1, but since the sentence is split up into individual words, it will stop anytime a word begins with capital X. Try entering a 5 word sentence with a capital X as the first character in the third word. You should get the first three words displayed, and the last two simply ignored when the program stops.

Try entering more than 24 characters to see what the program does. In an actual program, it is your responsibility to count characters and stop when the input buffer is full. You may be getting the feeling that a lot of

responsibility is placed on you when writing in C. It is, but you also get a lot of flexibility in the bargain too.

Line Input

It is possible to read a line of characters without dividing the line into individual words. The program [LINEIN.C] provides an example of how this can be accomplished.

```
#include "stdio.h"
void main( )
{
char line[25];
printf("Input a character string, up to 25 characters.\n");
gets(line)
printf("The string is -> %s\n", line);
printf("End of program.\n");
}
```

The gets function is a simple but dangerous function that achieves the purpose of reading in a line of characters. It is simple to use and works quite well except that it does not check to see if the user has entered more characters than can be stored by the character string. If the user does enter more than the designated number of characters then end result cannot be predicted. At the very least, other data values may be corrupted. At worst the program may crash. Clearly it is not desirable to allow users to enter data that could destroy the program. Although effective, the gets function should be used with caution.

Better Line Input

The C library does not provide a standard input function which validates the user's string input and checks that data remains within the length of the character string. The text file function fgets can be used with standard input to achieve this purpose. This function, however, will not be discussed here – it will be discussed in more detail in Module 816: File Input/Output in C.

A better line input function can be created from the character functions presented earlier. The program [GETLINE.C] illustrates how a function can read a line of text and simultaneously check that the data entry does not exceed the length of the character string.

The heart of the solution lies with the user-defined function called getline. This function is defined in a prototype on line 3 of the program. Its definition follows the main program. The getline function is passed two arguments. The first is the character string that will receive the data input. The character string, *s*, is an array variable, and therefore by definition, it is a pointer variable. No pointer notation is required. The second argument, *lim*, will determine the maximum number of characters that can be stored by the character string, *s*.

Module 814 Input and Output in C

```
#include "stdio.h"

void getline(char s[], int lim);

void main() {
    char line[80];
    printf("Please enter your line\n");
    getline(line, 30);
    printf("The line entered was %s\n\n",line);
}
/*===== */
/* GETLINE function definition */
/*===== */
void getline(char s[], int lim)
{
    /* local variables */
    int c, i;
    /* create a loop that terminates under three conditions */
    /* 1. lim - 1 characters have been entered */
    /* 2. an EOF character is entered at the keyboard */
    /* 3. a carriage return is entered at the keyboard */
    for (i=0; i<lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    /* if the limit has been reached, then skip over the */
    /* remaining characters. */
    while (c!='\n')
        c=getchar();
    /* Place the string termination character in the string. */
    s[i] = '\0';
}
```

The function consists essentially of two loops. The first collects characters from the standard input using the `getchar` function. This loop terminates when the limit has been reached or when the user terminates the string using an *end-of-file* character (Ctrl-Z on most systems), or a carriage return. The second loop will continue to read characters from the standard input – remember that DOS has collected a whole line of characters – until all characters have been removed. This step is necessary to prevent the next *getline* receiving part of this line as input. Finally, the function appends the C *end-of-string* character to the character string.

Compile and run this program. Test it to see if it correctly reads a specified number of characters from the standard input. Does it correctly handle carriage returns and extraneous input?

Exercise 2

Write a program that will prompt the user and accept as input an integer, a character, a string, and a floating point number.

Write a program to read in a character using a loop, and display the character in its normal char form. Also display it as a decimal number. Check for a dollar sign to use as the stop character. Use the `getch` form of input so it will print immediately. Hit some of the special keys, such as function keys, when you run the program for some surprises. You will get two inputs from the special keys, the first being a zero which is the indication to the system that a special key was hit.

Objective 3 After working through this module you should be able to use the standard output operations in C programs.

STANDARD OUTPUT OPERATIONS

Character output

The function

```
int putchar(int c)
```

is the fundamental output mechanism in C. `putchar(c)` puts the character *c* on the *standard output device*, which by default is the screen. Notice that `putchar` returns an integer value. This return value is used to test whether the output was successful or not. If the output was successful, then `putchar` returns the character that was displayed. If, for some reason, an error occurred during output then `putchar` will return the EOF value (-1 on most systems).

Like character input, character is not useful in most circumstances. Programmers usually are required to output numeric data, character strings and other structures. This kind of output is termed formatted output and the standard input/output library provides a powerful function that deals with this problem.

Formatted output

An example of formatted output [LOTTYPES.C] was provided in Module 813: Fundamental Programming Structures in C. The program is duplicated here for further study.

```
#include "stdio.h"
void main()
{
int a;           /* simple integer type           */
long int b;      /* long integer type            */
short int c;     /* short integer type          */
unsigned int d;  /* unsigned integer type       */
char e;         /* character type              */
float f;        /* floating point type         */
double g;       /* double precision floating point */

a = 1023;
b = 2222;
c = 123;
d = 1234;
e = 'X';
f = 3.14159;
g = 3.1415926535898;
printf("a = %d\n",a);           /* decimal output           */
printf("a = %o\n",a);           /* octal output             */
printf("a = %x\n",a);           /* hexadecimal output       */
printf("b = %ld\n",b);          /* decimal long output      */
printf("c = %d\n",c);           /* decimal short output     */
printf("d = %u\n",d);           /* unsigned output          */
printf("e = %c\n",e);           /* character output         */
}
```

```
printf("f = %f\n",f);      /* floating output          */
printf("g = %f\n",g);      /* double float output     */
printf("\n");
printf("a = %d\n",a);      /* simple int output       */
printf("a = %7d\n",a);     /* use a field width of 7  */
printf("a = %-7d\n",a);   /* left justify width = 7  */
printf("\n");
printf("f = %f\n",f);      /* simple float output     */
printf("f = %12f\n",f);    /* use field width of 12   */
printf("f = %12.3f\n",f);  /* use 3 decimal places    */
printf("f = %12.5f\n",f);  /* use 5 decimal places    */
printf("f = %-12.5f\n",f); /* left justify in field   */
}
```

Printing numbers

Let's return to the printf statements for a definition of how they work. Notice that they are all identical and that they all begin just like the printf statements we have seen before. The first difference occurs when we come to the % character. This is a special character that signals the output routine to stop copying characters to the output and do something different, namely output a variable. The % sign is used to signal the start of many different types of variables, but we will restrict ourselves to only one for this example. The character following the % sign is a d, which signals the output routine to get a decimal value and output it. Where the decimal value comes from will be covered shortly. After the d, we find the familiar \n, which is a signal to return the video 'carriage', and the closing quotation mark.

All of the characters between the quotation marks define the pattern of data to be output by this statement, and after the pattern, there is a comma followed by the variable name index. This is where the printf statement gets the decimal value which it will output because of the %d we saw earlier. We could add more %d output field descriptors within the brackets and more variables following the description to cause more data to be printed with one statement. Keep in mind, however, that it is important that the number of field descriptors and the number of variable definitions must be the same or the runtime system will get confused and probably quit with a runtime error.

Conversion characters

Here is a list of the conversion characters and the way they are used in the printf statement:

d	decimal notation
o	octal notation
x	hexadecimal notation
u	unsigned notation
c	character notation
s	string notation
f	floating point notation

Each of these is used following a percent sign to indicate the type of output conversion, and between those two characters, the following fields may be added.

- left justification in its field
- (n) a number specifying minimum field width
- . to separate n from m
- (m) significant fractional digits for a float
- l to indicate a "long"

These are all used in the examples which are included in the program presently displayed on your monitor, with the exception of the string notation which will be covered shortly. Note especially the variable field width specification in lines 33 to 36. Compile and run this program to see what effect the various fields have on the output.

Printing Character Strings

The `printf` can also be used to output character strings. The conversion character `s` is used to denote a string variable. The following statements indicate the different forms of output possible when the `printf` function is used with character strings. Assume that the variable `line` contains the string "Deakin University" which is 17 characters in length. The output expected is shown below each statement. The extent of each output is delimited by colons.

```
printf(":%s:", line);
:Deakin University:

printf(":%10s:", line);
:Deakin University:

printf(":%.10s:", line);
:Deakin Uni:

printf(":%-10s:", line);
:Deakin University:

printf(":%.20s:", line);
:Deakin University:

printf(":%-20s:", line);
:Deakin University :

printf(":%20.10s:", line);
: Deakin Uni:

printf(":%-20.10s:", line);
:Deakin Uni :
```

You should make sure that you can recognise and use each of these different forms of the `printf` function.

Notes

printf uses its first argument to decide how many arguments follow and what their types are. It will get confused, and you will get wrong answers, if there are not enough arguments or if they are the wrong type.

You now have the ability to display any of the data fields in the previous programs and it would be to your advantage to go back and see if you can display any of the fields anyway you desire.

Exercise 3

Consider the following two output statements:

```
printf(s);  
printf("%s", s);
```

If the character string *s* contains the string “Mary received 98% for her C test” the first statement will fail, whereas the second will work correctly produce the expected output. Can you explain why?

Write a program to determine if the statement

```
printf("%20.2f", num);
```

produces 97.77 or 97.78 when *num* is given the value of 97.7779. What can you conclude about the rounding or truncating features of the printf function?

Module 814 Input and Output in C

Objective 4 After working through this module you should be able to convert data values to formatted text using standard output operations.

DATA FORMATTING OPERATIONS

The next operation may seem a little strange at first, but you will probably see lots of uses for it as you gain experience. This program [INMEM.C] displays another type of I/O, one that never accesses the outside world, but stays in the computer.

```
void main( )
{
int numbers[5], result[5], index;
char line[80];
  numbers[0] = 74;
  numbers[1] = 18;
  numbers[2] = 33;
  numbers[3] = 30;
  numbers[4] = 97;
  sprintf(line, "%d %d %d %d %d\n", numbers[0], numbers[1], numbers[2],
           numbers[3], numbers[4]);
  printf("%s", line);
  sscanf(line, "%d %d %d %d %d", &result[4], &result[3],
         (result+2), (result+1), result);
  for (index = 0; index < 5; index++)
    printf("The final result is %d\n", result[index]);
}
```

First we define a few variables, then assign some values to the ones named numbers for illustrative purposes, and then use a sprintf function. The function acts just like a normal printf function except that instead of printing the line of output to a device, it prints the line of formatted output to a character string in memory. In this case the string goes to the string variable line, because that is the string name we inserted as the first argument in the sprintf function. The spaces after the 2nd %d were put there to illustrate that the next function will search properly across the line. We print the resulting string and find that the output is identical to what it would have been by using a printf instead of the sprintf in the first place. You will see that when you compile and run the program shortly.

Since the generated string is still in memory, we can now read it with the function sscanf. We tell the function in its first argument that line is the string to use for its input, and the remaining parts of the line are exactly what we would use if we were going to use the scanf function and read data from outside the computer. Note that it is essential that we use pointers to the data because we want to return data from a function. Just to illustrate that there are many ways to declare a pointer several methods are used, but all are pointers. The first two simply declare the address of the elements of the array, while the last three use the fact that result, without the accompanying subscript, is a pointer.

Just to keep it interesting, the values are read back in reverse order. Finally the values are displayed on the monitor.

It seems silly to read input data from within the computer but it does have a real purpose. It is possible to read data from an input device using any of the standard functions and then do a format conversion in memory. You could read in a line of data, look at a few significant characters, then use these formatted input routines to reduce the line of data to internal representation. That would certainly beat writing your own data formatting routines!

Exercise 4

Write a program that formats a floating point number into a string that contains the number in a currency format. For example, the number 23456.78 would be converted into the string "\$23,456.78".

Use the `sscanf` function to convert a date in the form of "dd/mm/yy" into the respective integers representing day, month and year.

Objective 5 After working through this module you should be able to state the significance of standard error output.

STANDARD ERROR OUTPUT

Sometimes it is desirable to redirect the output from the standard output device to a file. However, you may still want the error messages to go to the standard output device, in our case the monitor. This next function allows you to do that, and is shown in the next program [SPECIAL.C]:

```
#include "stdio.h"
void main( )
{
  int index;
  for (index = 0; index < 6; index++) {
    printf("This line goes to the standard output.\n");
    fprintf(stderr,"This line goes to the error device.\n");
  }
  exit(4);
  /* This can be tested with the DOS errorlevel command in a batch
  file.
      The number returned      is used as follows;
  IF ERRORLEVEL 4 GOTO FOUR
  (continue here if less than 4)
  .
  GOTO DONE
  :FOUR
  (continue here if 4 or greater)
  .
  :DONE          */
}
```

The program consists of a loop with two messages output, one to the standard output device and the other to the standard error device. The message to the standard error device is output with the function `fprintf` and includes the device name `stderr` as the first argument. Other than those two small changes, it is the same as our standard `printf` function. (You will see more of the `fprintf` function in the next section, but its operation fits in better as a part of this section.) Ignore the line with the `exit` for the moment, we will return to it.

Compile and run this program, and you will find 12 lines of output on the monitor. To see the difference, run the program again with redirected output to a file named `STUFF` by entering the following line at the DOS prompt;

```
special >stuff
```

More information about I/O redirection can be found in your DOS manual. This time you will only get the 6 lines output to the standard error device, and if you look in your directory, you will find the file named `STUFF` containing the other 6 lines, those to the standard output device. You can use I/O redirection with any of the programs we have

run so far, and as you may guess, you can also read from a file using I/O redirection.

Finally, let's look at 'exit(4)', the last statement in the program (before the multi-line comment). This simply exits the program and returns the value of 4 to DOS. Any number from 0 to 9 can be used in the parentheses for DOS communication. If you are operating in a BATCH file, this number can be tested with the ERRORLEVEL command.

Exercise 5

Test the output expected by the example program [SPECIAL.C]. Create the suggested batch file and execute the batch program to ensure that the error level value is being sent to DOS correctly.

Objective 6 After working through this module you should be able to use other input/output libraries, for example the Turbo C console library, to perform input and output operations.

Other input/output libraries

The ANSI standard for C does not define any text screen or graphics functions, mainly because the capabilities of diverse hardware environments preclude standardisation across a wide range of machines. However, Turbo C provides extensive screen (and graphics) support libraries for the PC environment. The use of these libraries, or other third-party library systems, can greatly enhance the user interface created by programmers. Because the libraries are non-standard, the programs that make use of them are not portable to other platforms. This, however, is often a low priority consideration since intensive screen control is often a must for most commercial programs.

The prototypes and header information for the text-screen handling functions are stored in the *conio.h* file. The major functions defined by this library will be discussed shortly. The prototypes and related information for the graphics system are held in the *graphics.h* file. The graphics system will not be discussed here. Interested students are directed to the Turbo C Reference Guide for further details about the use of this library.

A key factor in text screen manipulation is the *window*. A window is the active part of the screen within which output is displayed. A window can be as large as the entire screen (as it is by default), or it can be as small as you require. The upper-left corner of a window is located at the *x,y* position of 1,1. *x* refers to the column number and *y* refers to the row number. When a window occupies the entire screen the bottom-right corner of the window will be located at position 80,25.

You must be careful not to confuse a Turbo C window with windows produced by other systems, e.g. Microsoft Windows. Turbo C windows do not have borders, scroll bars, maximise or minimise buttons, or menus. A Turbo C window simply refers to an area of the screen that is to be used for output. Further, it is important to understand that most screen functions are window relative. This means that the position of the output is relative to the upper-left corner of the current window, **not** the screen.

CONIO Functions

The following definitions describe 12 functions that can be used to improve the standard output produced by C programs. Bear in mind

Description: The `gotoxy()` function sends the text screen cursor to the location specified by *x,y*. If either or both of the coordinates are invalid, no action takes place.

insline

Syntax: `void insline(void)`

Description: The `insline()` function inserts a blank line at the current cursor position, and all the lines below the cursor move down. The function only affects the active text window.

movetext

Syntax: `int movetext(int left, int top,
int right, int bottom,
int newleft, int newtop)`

Description: The `movetext()` function moves the portion of the screen defined by the rectangle *left,top* and *right,bottom* to the region of the screen that has its upper-left corner defined by *newleft,newtop*. The coordinates are screen coordinates, **not** window relative. The function returns 1 if the action was successful, 0 on failure.

puttext

Syntax: `int puttext(int left, int top,
int right, int bottom,
void *buf)`

Description: The `puttext()` function copies the text previously saved in the buffer pointed to by *buf* to the region defined by *left,top* and *right,bottom*. The coordinates are screen coordinates, **not** window relative. The function returns 1 if the action was successful, 0 on failure.

textattr

Syntax: `void clreol(int attr)`

Description: The `textattr()` function sets both the foreground and background colours in the text screen at one time. The value of *attr* represents an encoded form of the colour information. For example, to set the colours to be white text on a green background, *attr* would be defined as `GREEN*16 | WHITE`.

Exercise 6

Write a function that allows a text string to be written to the screen with the programmer controlling the coordinates of the first character, and the colour of the text and background.

Write a function that centres a given text string on the screen giving the programmer control over the line that the string will be drawn on, and the colour of the foreground and background. You may wish to investigate the LENGTH function in Turbo Pascal.

Write a function that writes a given text string vertically down the screen. Give the programmer control over then usual parameters. You will have to decide what to do if the string goes off the bottom of the screen.