

Input and Output Functions

Objectives:

Having read this section you should have a clearer idea of one of C's:

1. input functions, called scanf
2. output functions, called printf

On The Run:

Even with arithmetic you can't do very much other than write programs that are the equivalent of a pocket calculator. The real break through comes when you can read values into variables as the program runs. Notice the important words here: "as the program runs". You can already store values in variables using assignment. That is:

```
a=100;
```

stores 100 in the variable a each time you run the program, no matter what you do. Without some sort of input command every program would produce exactly the same result every time it was run. This would certainly make debugging easy! But in practice, of course, we need programs to do different jobs each time they are run. There are a number of different C input commands, the most useful of which is the scanf command. To read a single integer value into the variable called a you would use:

```
scanf("%d",&a);
```

For the moment don't worry about what the %d or the >&a means - concentrate on the difference between this and:

```
a=100;
```

When the program reaches the scanf statement it pauses to give the user time to type something on the keyboard and continues only when users press <Enter>, or <Return>, to signal that he, or she, has finished entering the value. Then the program continues with the new value stored in a. In this way, each time the program is run the user gets a chance to type in a different value to the variable and the program also gets the chance to produce a different result!

The final missing piece in the jigsaw is using the printf function, the one we have already used to print "Hello World", to print the value currently being stored in a variable. To display the value stored in the variable a you would use:

```
printf("The value stored in a is %d",a);
```

The %d, both in the case of scanf and printf, simply lets the compiler know that the value being read in, or printed out, is a decimal integer - that is, a few digits but no decimal point.

Note: the scanf function does not prompt for an input. You should get in the habit of always using a printf function, informing the user of the program what they should type, before a scanf function.

Input and Output Functions in More Detail:

One of the advantages of C is that essentially it is a small language. This means that you can write a complete description of the language in a few pages. It doesn't have many keywords or data types for that matter. What makes C so powerful is the way that these low-level facilities can be put together to make higher level facilities.

The only problem with this is that C programmers have a tendency to reinvent the wheel each time they want to go for a ride. It is also possible to write C programs in a variety of styles which depend on the particular tricks and devices that a programmer chooses to use. Even after writing C for a long time you will still find the occasionally construction which makes you think, "I never thought of that!" or, "what is that doing?"

One attempt to make C a more uniform language is the provision of standard libraries of functions that perform common tasks. We say standard but until the ANSI committee actually produced a standard there was, and still is, some variation in what the standard libraries contained and exactly how the functions

worked. Having said that we had better rush in quickly with the reassurance that in practice the situation isn't that bad and most of the functions that are used frequently really are standard on all implementations. In particular the I/O functions vary very little.

It is now time to look at exactly how `scanf` and `printf` work and what they can do - you might be surprised at just how complex they really are!

The original C specification did not include commands for input and output. Instead the compiler writers were supposed to implement library functions to suit their machines. In practice all chose to implement `printf` and `scanf` and after a while C programmers started to think of them as if these functions were I/O keywords! It sometimes helps to remember that they are functions on a par with any other functions you may care to define. If you want to you can provide your own implementations of `printf` or `scanf` or any of the other standard functions - we'll discover how later.

`printf`:

The `printf` (and `scanf`) functions do differ from the sort of functions that you will create for yourself in that they can take a variable number of parameters. In the case of `printf` the first parameter is always a string (c.f. "Hello World") but after that you can include as many parameters of any type that you want to. That is, the `printf` function is usually of the form:

```
printf(string,variable,variable,variable...)
```

where the ... means you can carry on writing a list of variables separated by commas as long as you want to. The string is all-important because it specifies the type of each variable in the list and how you want it printed. The string is usually called the control string or the format string. The way that this works is that `printf` scans the string from left to right and prints on the screen, or any suitable output device, any characters it encounters - except when it reaches a % character. The % character is a signal that what follows it is a specification for how the next variable in the list of variables should be printed. `printf` uses this information to convert and format the value that was passed to the function by the variable and then moves on to process the rest of the control string and any more variables it might specify. For example:

```
printf("Hello World");
```

only has a control string and, as this contains no % characters it results in Hello World being displayed and doesn't need to display any variable values. The specifier %d means convert the next value to a signed decimal integer and so:

```
printf("Total = %d",total);
```

will print Total = and then the value passed by >total as a decimal integer.

If you are familiar with other programming languages then you may feel happy about the `printf` function because something like:

```
printf("Total = %d",total);
```

looks like the sort of output command you might have used before. For example, in BASIC you would write:

```
PRINT "Total = ",total
```

but the C view of output is at a lower level than you might expect. The %d isn't just a format specifier, it is a conversion specifier. It indicates the data type of the variable to be printed and how that data type should be converted to the characters that appear on the screen. That is %d says that the next value to be printed is a signed integer value (i.e. a value that would be stored in a standard int variable) and this should be converted into a sequence of characters (i.e. digits) representing the value in decimal. If by some accident the variable that you are trying to display happens to be a float or a double then you will still see a value displayed - but it will not correspond to the actual value of the float or double.

The reason for this is twofold.

1. The first difference is that an int uses two bytes to store its value, while a float uses four and a double uses eight. If you try to display a float or a double using %d then only the first two bytes of the value are actually used.
2. The second problem is that even if there wasn't a size difference ints, floats and doubles use a different binary representation and %d expects the bit pattern to be a simple signed

binary integer.

This is all a bit technical, but that's in the nature of C. You can ignore these details as long as you remember two important facts:

1. The specifier following % indicates the type of variable to be displayed as well as the format in which that the value should be displayed;
2. If you use a specifier with the wrong type of variable then you will see some strange things on the screen and the error often propagates to other items in the printf list.

If this seems complicated then I would agree but I should also point out that the benefit is being able to treat what is stored in a variable in a more flexible way than other languages allow. Other languages never let on to the programmer that what is in fact stored in a variable is a bit pattern, not the decimal value that appears to be stored there when you use a printf (or whatever) statement. Of course whether you view this as an advantage depends on what you are trying to do. It certainly brings you closer to the way the machine works.

You can also add an 'l' in front of a specifier to mean a long form of the variable type and h to indicate a short form (long and short will be covered later in this course). For example, %ld means a long integer variable (usually four bytes) and %hd means short int. Notice that there is no distinction between a four-byte float and an eight-byte double. The reason is that a float is automatically converted to a double precision value when passed to printf - so the two can be treated in the same way. (In pre-ANSI all floats were converted to double when passed to a function but this is no longer true.) The only real problem that this poses is how to print the value of a pointer? The answer is that you can use %x to see the address in hex or %o to see the address in octal. Notice that the value printed is the segment offset and not the absolute address - to understand what we are going on about you need to know something about the structure of your processor.

The % Format Specifiers:

The % specifiers that you can use in ANSI C are:

Usual variable type	Display
%c char	single character
%d (%i) int	signed integer
%e (%E) float or double	exponential format
%f float or double	signed decimal
%g (%G) float or double	use %f or %e as required
%o int	unsigned octal value
%p pointer	address stored in pointer
%s array of char	sequence of characters
%u int	unsigned decimal
%x (%X) int	unsigned hex value

Formatting Your Output:

The type conversion specifier only does what you ask of it - it convert a given bit pattern into a sequence of characters that a human can read. If you want to format the characters then you need to know a little more about the printf function's control string.

Each specifier can be preceded by a modifier which determines how the value will be printed. The most general modifier is of the form:

flag width.precision

The flag can be any of:

flag	meaning
-	left justify
+	always display sign
space	display space if there is no sign
0	pad with leading zeros
#	use alternate form of specifier

The width specifies the number of characters used in total to display the value and precision indicates the number of characters

used after the decimal point.

For example, %10.3f will display the float using ten characters with three digits after the decimal point. Notice that the ten characters includes the decimal point, and a - sign if there is one.

If the value needs more space than the width specifies then the additional space is used - width specifies the smallest space that will be used to display the value. (This is quiet reassuring, you won't be the first programmer whose program takes hours to run but the output results can't be viewed because the wrong format width has been specified!)

The specifier %-10d will display an int left justified in a ten character space. The specifier %+5d will display an int using the next five character locations and will add a + or - sign to the value.

The only complexity is the use of the # modifier. What this does depends on which type of format it is used with:

%#o adds a leading 0 to the octal value
%#x adds a leading 0x to the hex value
%#f or
%#e ensures decimal point is printed
%#g displays trailing zeros

Strings will be discussed later but for now remember: if you print a string using the %s specifier then all of the characters stored in the array up to the first null will be printed. If you use a width specifier then the string will be right justified within the space. If you include a precision specifier then only that number of characters will be printed.

For example:

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

will print Hello,

```
printf("%25s ,Hello")
```

will print 25 characters with Hello right justified and

```
printf("%25.3s,Hello")
```

will print Hello right justified in a group of 25 spaces.

Also notice that it is fine to pass a constant value to printf as in printf("%s,Hello").

Finally there are the control codes:

\b backspace
\f formfeed
\n new line
\r carriage return
\t horizontal tab
\' single quote
\0 null

If you include any of these in the control string then the corresponding ASCII control code is sent to the screen, or output device, which should produce the effect listed. In most cases you only need to remember \n for new line.

scanf:

Now that we have mastered the intricacies of printf you should find scanf very easy. The scanf function works in much the same way as the printf. That is it has the general form:

```
scanf(control string,variable,variable,...)
```

In this case the control string specifies how strings of characters, usually typed on the keyboard, should be converted into values and stored in the listed variables. However there are a number of important differences as well as similarities between scanf and printf.

The most obvious is that scanf has to change the values stored in the parts of computers memory that is associated with parameters (variables).

To understand this fully you will have to wait until we have covered functions in more detail. But, just for now, bare with us when we say to do this the scanf function has to have the addresses of the variables rather than just their values. This means that simple variables have to be passed with a preceding >&. (Note for future reference: There is no need to do this for strings stored in arrays because the array name is already a pointer.)

The second difference is that the control string has some extra items to cope with the problems of reading data in. However, all of the conversion specifiers listed in connection with printf can be used with scanf.

The rule is that scanf processes the control string from left to right and each time it reaches a specifier it tries to interpret what has been typed as a value. If you input multiple values then these are assumed to be separated by white space - i.e. spaces, newline or tabs. This means you can type:

3 4 5

or

3
4
5

and it doesn't matter how many spaces are included between items. For example:

```
scanf("%d %d",&i,&j);
```

will read in two integer values into i and j. The integer values can be typed on the same line or on different lines as long as there is at least one white space character between them.

The only exception to this rule is the %c specifier which always reads in the next character typed no matter what it is. You can also use a width modifier in scanf. In this case its effect is to limit the number of characters accepted to the width.

For example:

```
scanf("%lOd",&i)
```

would use at most the first ten digits typed as the new value for i.

There is one main problem with scanf function which can make it unreliable in certain cases. The reason being is that scanf tends to ignore white spaces, i.e. the space character. If you require your input to contain spaces this can cause a problem. Therefore for string data input the function getstr() may well be more reliable as it records spaces in the input text and treats them as an ordinary characters.

Custom Libraries:

If you think printf and scanf don't seem enough to do the sort of job that any modern programmer expects to do, you would be right. In the early days being able to print a line at a time was fine but today we expect to be able to print anywhere on the screen at any time.

The point is that as far as standard C goes simple I/O devices are stream-oriented - that is you send or get a stream of characters without any notion of being able to move the current position in the stream. If you want to move backwards and forwards through the data then you need to use a direct access file. In more simple terms, C doesn't have a Tab(X,Y) or Locate(X,Y) function or command which moves the cursor to the specified location! How are you ever going to write your latest block buster game, let alone build your sophisticated input screens?

Well you don't have to worry too much because although C may not define them as standard, all C implementations come with an extensive graphics/text function library that allows you to do all of this and more. Such a library isn't standard, however the principles are always the same. The Borland and Microsoft offerings are usually considered as the two facto standards.

Summing It Up:

Now that we have arithmetic, a way of reading values in and a way of displaying them, it's possible to write a slightly more interesting program than "Hello World". Not much more interesting, it's true, but what do you expect with two instructions and some arithmetic?

Let's write a program that adds two numbers together and prints the result. (I told you it wasn't that much more interesting!) Of course, if you want to work out something else like fahrenheit to centigrade, inches to centimetres or the size of your bank balance, then that's up to you - the principle is the same.

The program is a bit more complicated than you might expect, but only because of the need to let the user know what is happening:

```
#include <stdio.h>
main()
{
    int a,b,c;
    printf("\nThe first number is ");
    scanf("%d",&a);
    printf("The second number is ");
    scanf("%d",&b);
    c=a+b;
    printf("The answer is %d \n",c);
}
```

The first instruction declares three integer variables: a, b and c. The first two printf statements simply display message on the screen asking the user for the values. The scanf functions then read in the values from the keyboard into a and b. These are added together and the result in c is displayed on the screen with a suitable message. Notice the way that you can include a message in the printf statement along with the value.

Type the program in, compile it and link it and the result should be your first interactive program. Try changing it so that it works out something a little more adventurous. Try changing the messages as well. All you have to remember is that you cannot store values or work out results greater than the range of an integer variable or with a fractional part.