EECS Cheatsheet

C was originally developed in the 1970s by Dennis Ritchie at Bell Telephone Laboratories, Inc. It is a **general-purpose, structured programming language**.

Each function must contain:

1. A function *heading*, which consists of the function name, followed by an optional list of *arguments*, enclosed in parentheses.
2. A list of argument *declarations*, if arguments are included in the heading.
3. A *compound statement*, which comprises the remainder of the function.

Each compound statement is enclosed within a pair of braces, i.e., `{ }`. The braces may contain one or more elementary statements (called *expression statements*) and other compound statements. Thus compound statements may be nested, one within another. Each expression statement must end with a semicolon (`;`).

Comments (remarks) may appear anywhere within a program, as long as they are placed within the delimiters `/*` and `*/` (e.g., `/* this is a comment */`). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

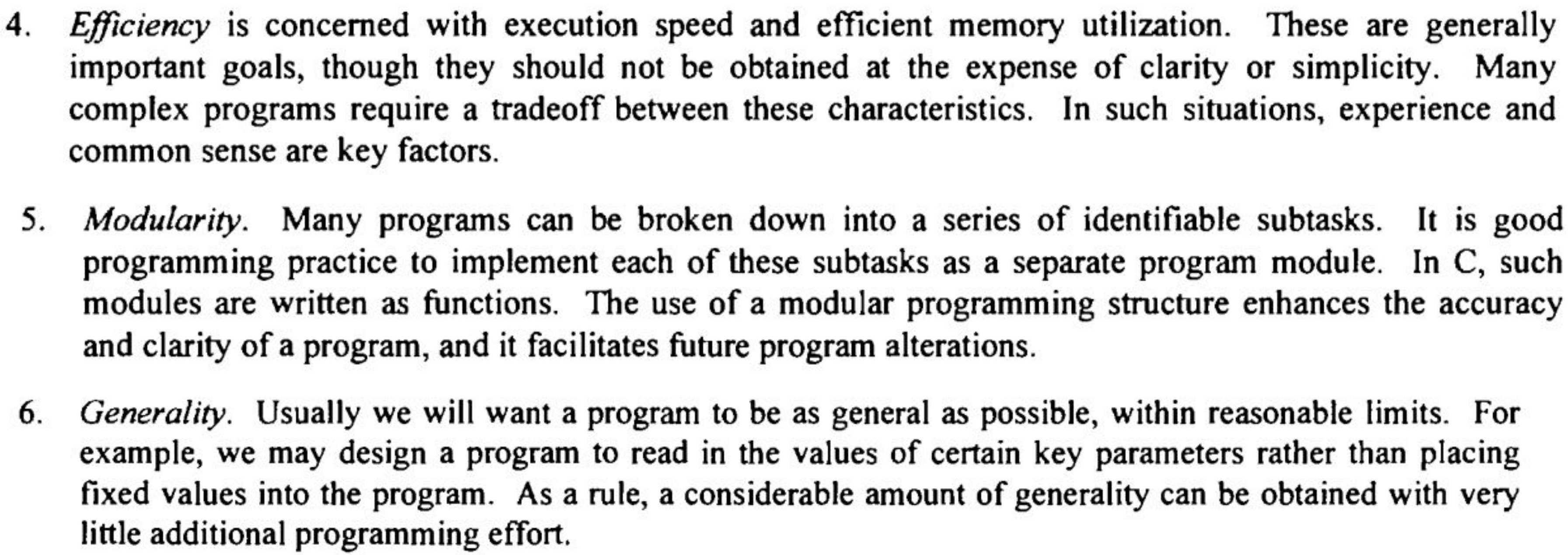
Desirable Program Characteristics

1. *Integrity*. This refers to the accuracy of the calculations. It should be clear that all other program enhancements will be meaningless if the calculations are not carried out correctly. Thus, the integrity of the calculations is an absolute necessity in any computer program.

2. *Clarity* refers to the overall readability of the program, with particular emphasis on its underlying logic. If a program is clearly written, it should be possible for another programmer to follow the program logic without undue effort. It should also be possible for the original author to follow his or her own program after being away from the program for an extended period of time. One of the objectives in the design of C is the development of clear, readable programs through an orderly and disciplined approach to programming.

3. *Simplicity*. The clarity and accuracy of a program are usually enhanced by keeping things as simple as possible, consistent with the overall program objectives. In fact, it may be desirable to sacrifice a certain amount of computational efficiency in order to maintain a relatively simple, straightforward program structure.

Desirable Program Characteristics

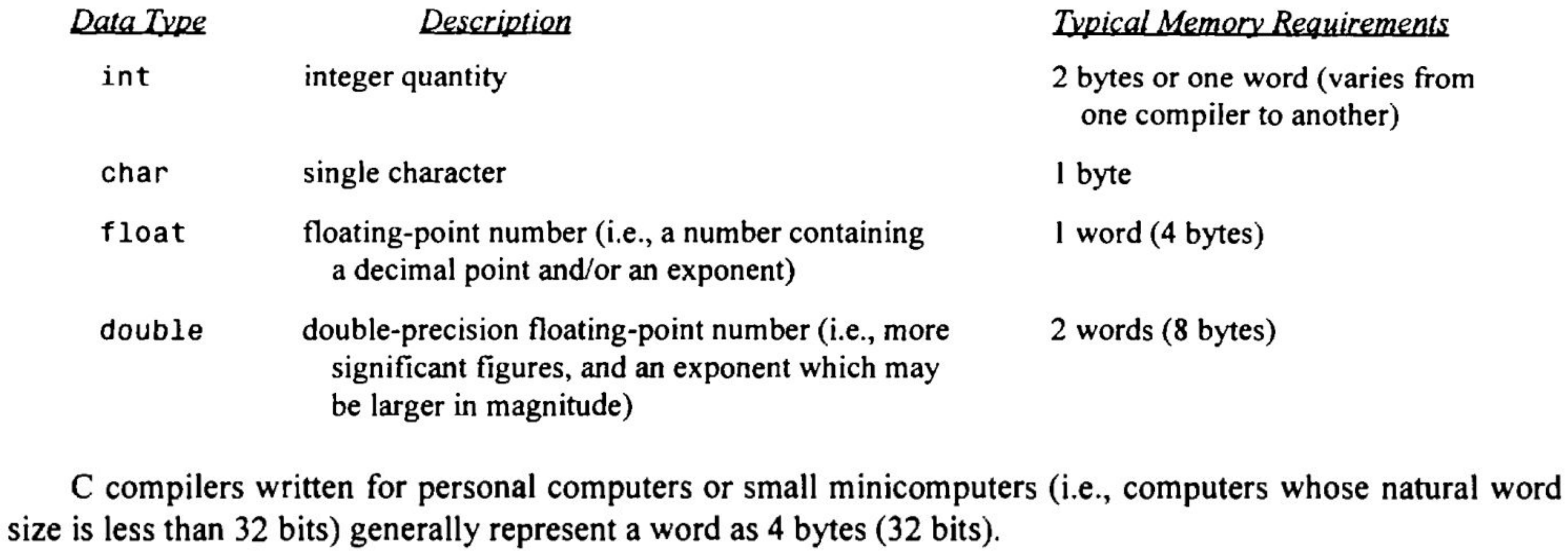


4. *Efficiency* is concerned with execution speed and efficient memory utilization. These are generally important goals, though they should not be obtained at the expense of clarity or simplicity. Many complex programs require a tradeoff between these characteristics. In such situations, experience and common sense are key factors.

5. *Modularity*. Many programs can be broken down into a series of identifiable subtasks. It is good programming practice to implement each of these subtasks as a separate program module. In C, such modules are written as functions. The use of a modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations.

6. *Generality*. Usually we will want a program to be as general as possible, within reasonable limits. For example, we may design a program to read in the values of certain key parameters rather than placing fixed values into the program. As a rule, a considerable amount of generality can be obtained with very little additional programming effort.

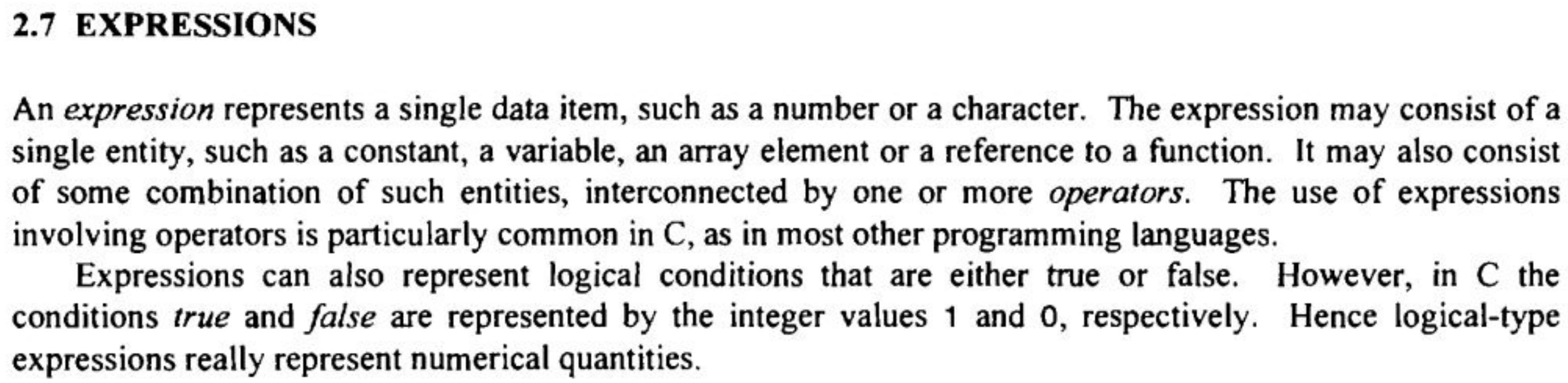
Memory Space



Data Type	*Description*	*Typical Memory Requirements*
`int`	integer quantity	2 bytes or one word (varies from one compiler to another)
`char`	single character	1 byte
`float`	floating-point number (i.e., a number containing a decimal point and/or an exponent)	1 word (4 bytes)
`double`	double-precision floating-point number (i.e., more significant figures, and an exponent which may be larger in magnitude)	2 words (8 bytes)

C compilers written for personal computers or small minicomputers (i.e., computers whose natural word size is less than 32 bits) generally represent a word as 4 bytes (32 bits).

Expressions

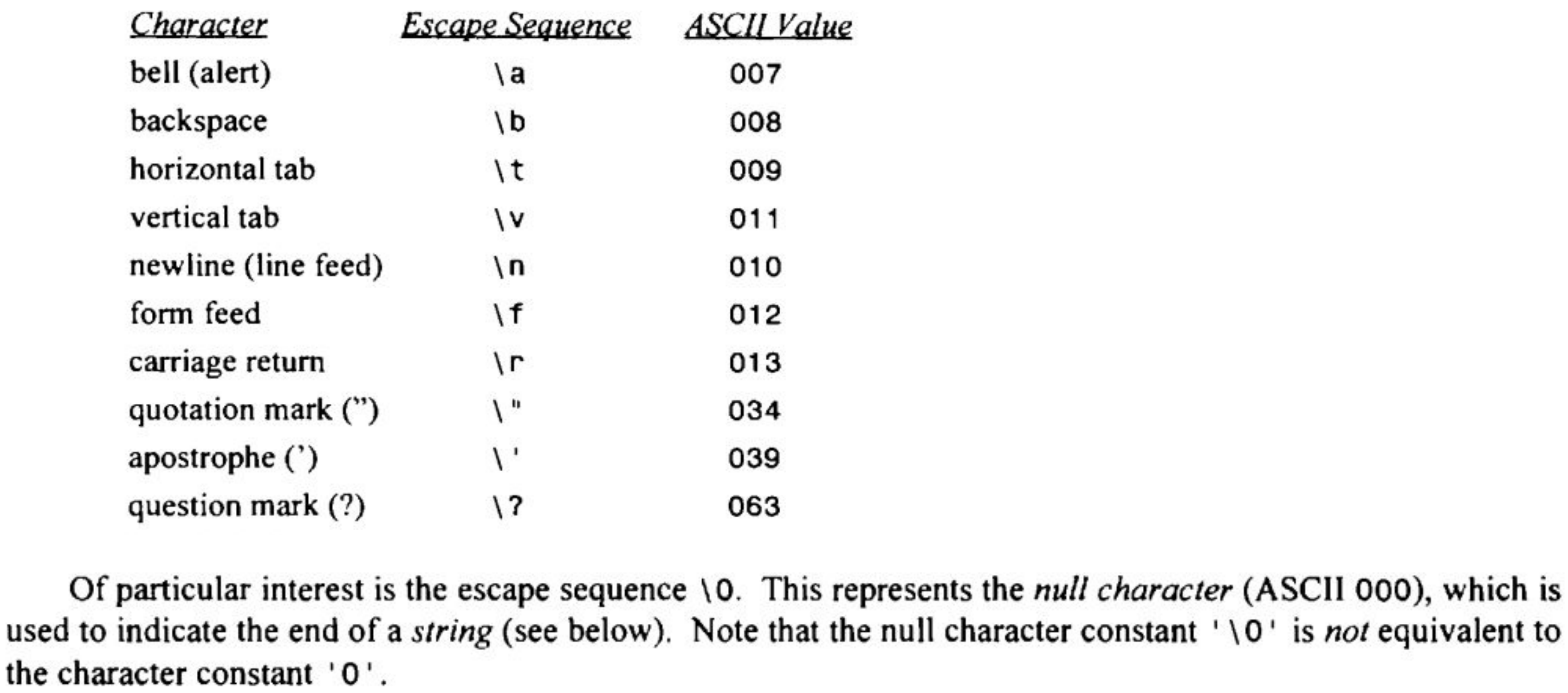


2.7 EXPRESSIONS

An *expression* represents a single data item, such as a number or a character. The expression may consist of a single entity, such as a constant, a variable, an array element or a reference to a function. It may also consist of some combination of such entities, interconnected by one or more *operators*. The use of expressions involving operators is particularly common in C, as in most other programming languages.

Expressions can also represent logical conditions that are either true or false. However, in C the conditions *true* and *false* are represented by the integer values 1 and 0, respectively. Hence logical-type expressions really represent numerical quantities.

Escape Sequences



Character	*Escape Sequence*	*ASCII Value*
bell (alert)	`\a`	007
backspace	`\b`	008
horizontal tab	`\t`	009
vertical tab	`\v`	011
newline (line feed)	`\n`	010
form feed	`\f`	012
carriage return	`\r`	013
quotation mark (")	`\"`	034
apostrophe (')	`\'`	039
question mark (?)	`\?`	063

Of particular interest is the escape sequence `\0`. This represents the *null character* (ASCII 000), which is used to indicate the end of a *string* (see below). Note that the null character constant `'\0'` is *not* equivalent to the character constant `'0'`.

Table 3-1 Operator Precedence Groups

Operator category	*Operators*	*Associativity*
unary operators	`-` `++` `--` `!` `sizeof` `(`*`type`*`)`	R → L
arithmetic multiply, divide and remainder	`*` `/` `%`	L → R
arithmetic add and subtract	`+` `-`	L → R
relational operators	`<` `<=` `>` `>=`	L → R
equality operators	`==` `!=`	L → R
logical *and*	`&&`	L → R
logical *or*	`||`	L → R
conditional operator	`? :`	R → L
assignment operators	`=` `+=` `-=` `*=` `/=` `%=`	R → L

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

An assignment operator is used for assigning a value to a variable. The most common assignment operator is `=`

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Table 3-2 Some Commonly Used Library Functions

Function	*Type*	*Purpose*
`abs(i)`	int	Return the absolute value of `i`.
`ceil(d)`	double	Round up to the next integer value (the smallest integer that is greater than or equal to `d`).
`cos(d)`	double	Return the cosine of `d`.
`cosh(d)`	double	Return the hyperbolic cosine of `d`.
`exp(d)`	double	Raise *e* to the power `d` ($e = 2.7182818 \cdots$ is the base of the natural (Naperian) system of logarithms).
`fabs(d)`	double	Return the absolute value of `d`.
`floor(d)`	double	Round down to the next integer value (the largest integer that does not exceed `d`).
`fmod(d1,d2)`	double	Return the remainder (i.e., the noninteger part of the quotient) of `d1/d2`, with same sign as `d1`.
`getchar()`	int	Enter a character from the standard input device.
`log(d)`	double	Return the natural logarithm of `d`.

`pow(d1,d2)`	double	Return `d1` raised to the `d2` power.
`printf(...)`	int	Send data items to the standard output device (arguments are complicated – see Chap. 4).
`putchar(c)`	int	Send a character to the standard output device.
`rand()`	int	Return a random positive integer.
`sin(d)`	double	Return the sine of `d`.
`sqrt(d)`	double	Return the square root of `d`.
`srand(u)`	void	Initialize the random number generator.
`scanf(...)`	int	Enter data items from the standard input device (arguments are complicated – see Chap. 4).
`tan(d)`	double	Return the tangent of `d`.
`toascii(c)`	int	Convert value of argument to ASCII.
`tolower(c)`	int	Convert letter to lowercase.
`toupper(c)`	int	Convert letter to uppercase.

Note: *Type* refers to the data type of the quantity that is returned by the function.

`c` denotes a character-type argument

`i` denotes an integer argument

`d` denotes a double-precision argument

`u` denotes an unsigned integer argument

Library Functions

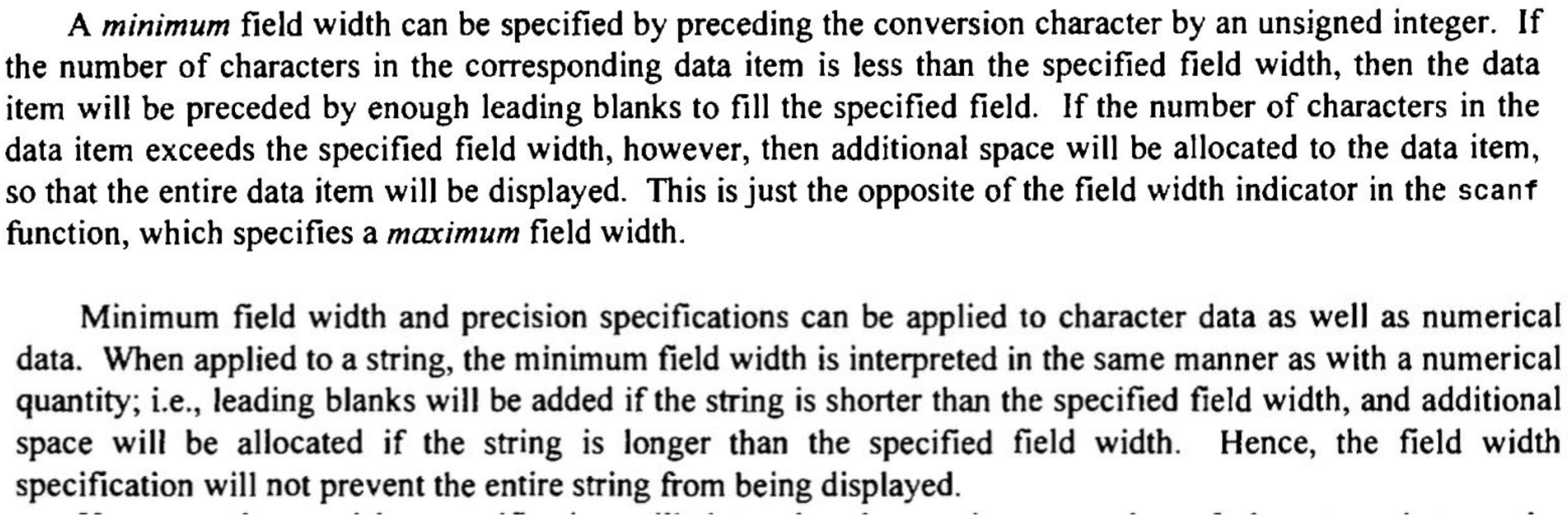
This program contains three library functions: `getchar`, `toupper` and `putchar`. The first two functions each return a single character (`getchar` returns a character that is entered from the keyboard, and `toupper` returns the uppercase equivalent of its argument). The last function (`putchar`) causes the value of the argument to be displayed. Notice that the last two functions each have one argument but the first function does not have any arguments, as indicated by the empty parentheses.

Also, notice the preprocessor statements `#include <stdio.h>` and `#include <ctype.h>`, which appear at the start of the program. These statements cause the contents of the files `stdio.h` and `ctype.h` to be inserted into the program the compilation process begins. The information contained in these files is essential for the proper functioning of the library functions `getchar`, `putchar` and `toupper`.

Table 4-1 Commonly Used Conversion Characters for Data Input

Conversion Character	*Meaning*
`c`	data item is a single character
`d`	data item is a decimal integer
`e`	data item is a floating-point value
`f`	data item is a floating-point value
`g`	data item is a floating-point value
`h`	data item is a short integer
`i`	data item is a decimal, hexadecimal or octal integer
`o`	data item is an octal integer
`s`	data item is a string followed by a whitespace character (the null character `\0` will automatically be added at the end)
`u`	data item is an unsigned decimal integer
`x`	data item is a hexadecimal integer
`[. . .]`	data item is a string which may include whitespace characters (see explanation below)

Minimum field



A *minimum* field width can be specified by preceding the conversion character by an unsigned integer. If the number of characters in the corresponding data item is less than the specified field width, then the data item will be preceded by enough leading blanks to fill the specified field. If the number of characters in the data item exceeds the specified field width, however, then additional space will be allocated to the data item, so that the entire data item will be displayed. This is just the opposite of the field width indicator in the `scanf` function, which specifies a *maximum* field width.

Minimum field width and precision specifications can be applied to character data as well as numerical data. When applied to a string, the minimum field width is interpreted in the same manner as with a numerical quantity; i.e., leading blanks will be added if the string is shorter than the specified field width, and additional space will be allocated if the string is longer than the specified field width. Hence, the field width specification will not prevent the entire string from being displayed.

Flag

In addition to the field width, the precision and the conversion character, each character group within the control string can include a *flag*, which affects the appearance of the output. The flag must be placed immediately after the percent sign (%). Some compilers allow two or more flags to appear consecutively, within the same character group. The more commonly used flags are listed in Table 4-3.

The `gets` and `puts` functions offer simple alternatives to the use of `scanf` and `printf` for reading and displaying strings, as illustrated in the following example.

Table 4-3 Commonly Used Flags

Flag	*Meaning*
–	Data item is left justified within the field (blank spaces required to fill the minimum field width will be added *after* the data item rather than *before* the data item).
+	A sign (either + or –) will precede each signed numerical data item. Without this flag, only negative data items are preceded by a sign.
`0`	Causes leading zeros to appear instead of leading blanks. Applies only to data items that are right justified within a field whose minimum size is larger than the data item. (*Note:* Some compilers consider the zero flag to be a part of the field width specification rather than an actual flag. This assures that the `0` is processed last, if multiple flags are present.)
' ' (*blank space*)	A blank space will precede each positive signed numerical data item. This flag is overridden by the + flag if both are present.
# (*with* `o`- *and* `x`-*type conversion*)	Causes octal and hexadecimal data items to be preceded by `0` and `0x`, respectively.
# (*with* `e`-, `f`- *and* `g`-*type conversion*)	Causes a decimal point to be present in all floating-point numbers, even if the data item is a whole number. Also prevents the truncation of trailing zeros in `g`-type conversion.