جامعة بيرزيت

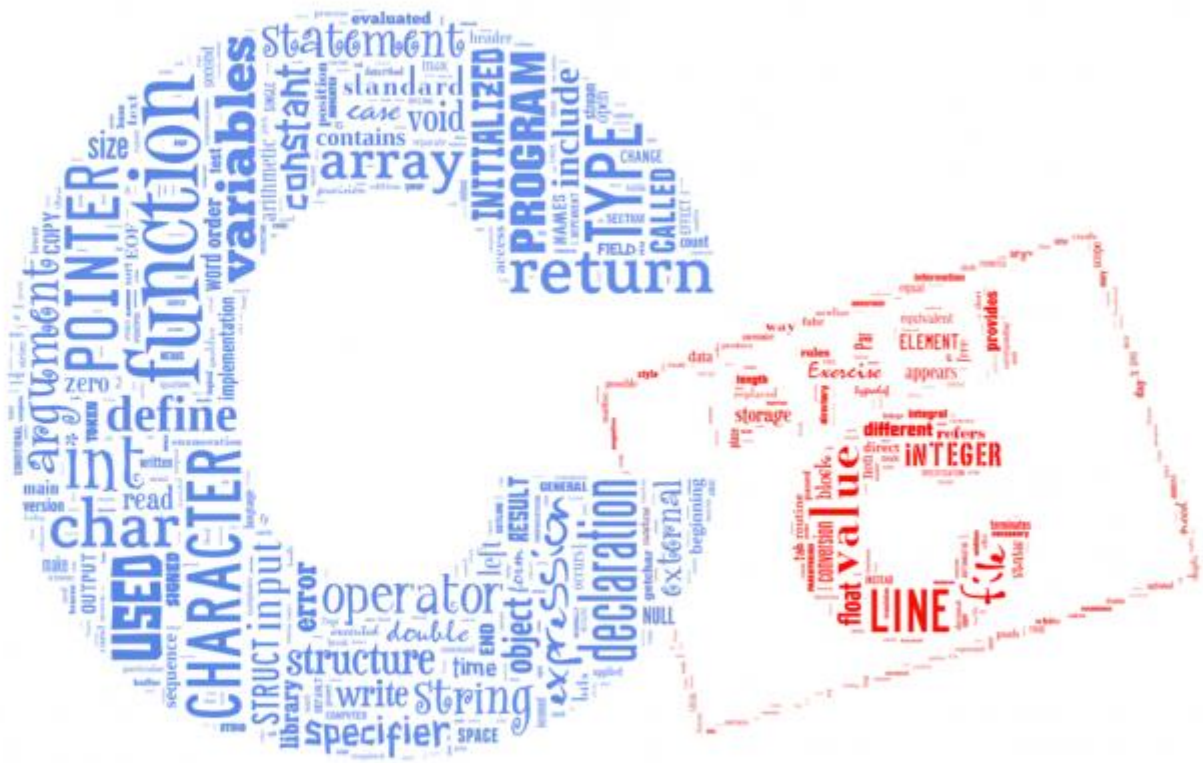
BIRZEIT UNIVERSITY

Faculty of Engineering and Technology كلية الهندسة والتكنولوجيا

Computer Science Department دائرة علم الحاسوب

COMP142/ COMP133/ COMP230/ COMP132

Introduction to Computer and Programming



Compiled and Prepared by:

Dr. Majdi M. Mafarja

Chapter 1: Introduction

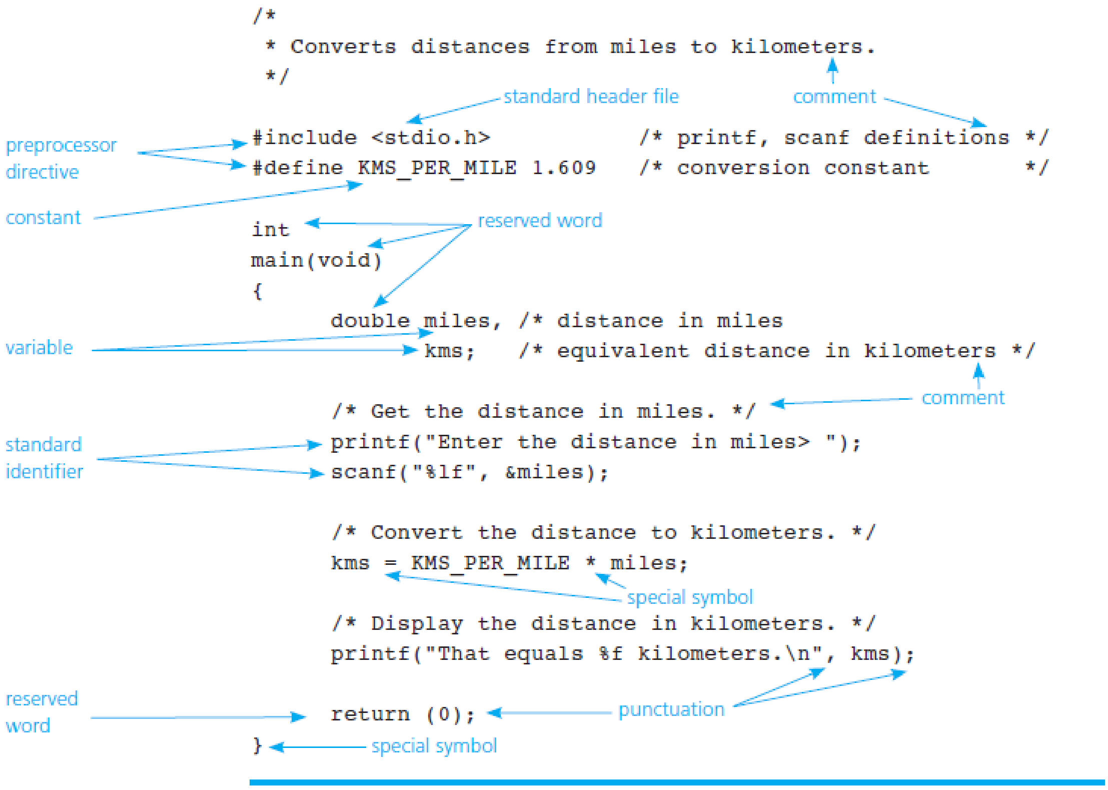
C Language Elements:

C Language is a sequential programming language, i.e. it compiles the program line by line sequentially.

A C program consists of one or more functions or code modules, which are groups of C statements to be executed in a given order. Each C program must contain a ***main()*** function. This is the first function called when the program starts to run.

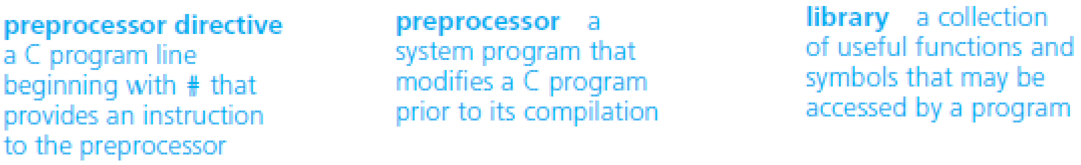
As the figure below, the C program has two parts: preprocessor directives and the main function.

FIGURE 2.1 C Language Elements in Miles-to-Kilometers Conversion Program



```
/*
 * Converts distances from miles to kilometers.
 */

#include <stdio.h>           /* printf, scanf definitions */
#define KMS_PER_MILE 1.609   /* conversion constant       */

int
main(void)
{
      double miles, /* distance in miles
             kms;   /* equivalent distance in kilometers */

      /* Get the distance in miles. */
      printf("Enter the distance in miles> ");
      scanf("%lf", &miles);

      /* Convert the distance to kilometers. */
      kms = KMS_PER_MILE * miles;

      /* Display the distance in kilometers. */
      printf("That equals %f kilometers.\n", kms);

      return (0);
}
```

standard header file
comment
preprocessor directive
constant
reserved word
variable
comment
standard identifier
special symbol
reserved word
punctuation
special symbol



preprocessor directive a C program line beginning with # that provides an instruction to the preprocessor

preprocessor a system program that modifies a C program prior to its compilation

library a collection of useful functions and symbols that may be accessed by a program

The "Hello World" Program

Consider first a simple C program which simply prints a line of text to the computer screen. This is traditionally the first C program you will see and is commonly called the "Hello World" program for obvious reasons.

```
#include <stdio.h>

void main()
{
      /* This is how comments are implemented in C
                  to comment out a block of text */
      // or like this for a single line comment

      printf( "Hello World\n" ) ;
}
```

As you can see this program consists of just one function, the mandatory *main* function.

- The parentheses, ( ), after the word main indicate that this is a function.
- The curly braces, { }, are used to denote a block of code (the start and end of the main function)

To place a comment (a statement that will not be compiled) you can use:

- a /* ... */ pair in the case of a block comment (more than one line of code) or
- a double forward slash, //, may be used to comment out the remains of a single line of test.

The line

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

- This statement is used to print an output on the screen. It must be terminated by a semicolon.
- The characters \n prints a newline on the screen.
  `printf` is a function that is contained in the `stdio` library which included in your program by the statement: `#include <stdio.h>`

NB: C is case sensitive i.e. `printf()` and `Printf()` would be regarded as two different functions.

Reserved Words:

A reserved word is a word that has a special meaning in C.

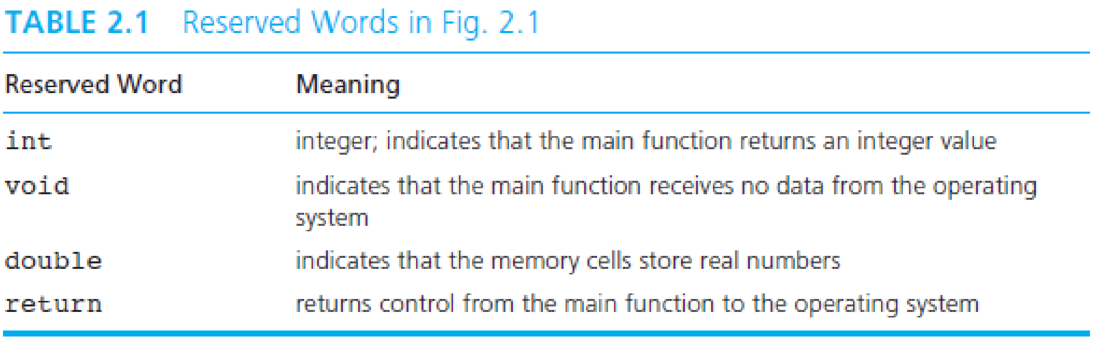


TABLE 2.1 Reserved Words in Fig. 2.1

Reserved Word	Meaning
`int`	integer; indicates that the main function returns an integer value
`void`	indicates that the main function receives no data from the operating system
`double`	indicates that the memory cells store real numbers
`return`	returns control from the main function to the operating system

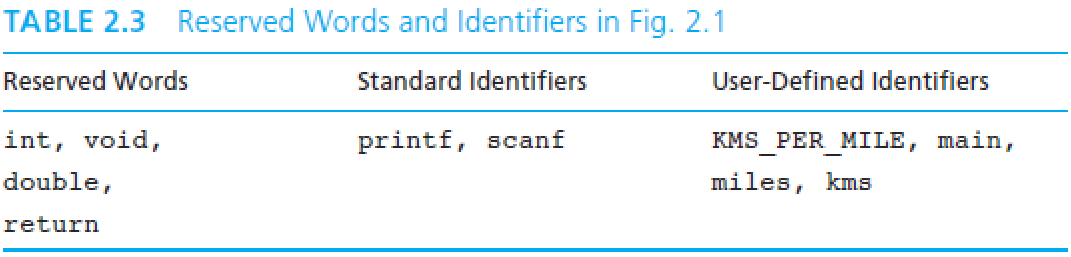


TABLE 2.3 Reserved Words and Identifiers in Fig. 2.1

Reserved Words	Standard Identifiers	User-Defined Identifiers

`int, void,`
`double,`
`return`
	`printf, scanf`	`KMS_PER_MILE, main,`
		`miles, kms`

User-Defined Identifiers (Variables):

In order to make the program flexible a useful, the programmer can identify (يعرّف) his own identifiers to name memory cells that will hold real life quantities or data e.g. a person’s name, age, height, bank balance, etc.

Syntax: **`type variable_name (or variable-list);`**

Basic Data Types

There are five basic data types char, int, float, double, and void. All other data types in C are based on these.

char	1 byte (8 bits) with range -128 to 127
int	4 bytes with range -2,147,483,648 **to** 2,147,483,647
float	4 bytes with range 10^{-38} to 10^{38} with 7 digits of precision
double	8 bytes with range 10^{-308} to 10^{308} with 15 digits of precision
void	generic pointer, used to indicate no function parameters etc.

For Example :-

```
int i ; // a memory cell is reserved to hold an integer value
char a, b, ch ; //three character variables are defined
```

Variable Names

Names of variables and functions in C are called identifiers and are case sensitive. When defining a variable you have to follow the following rules:

1. An identifier must consist only of letters, digits, and underscores (NO special characters like: +*&^%#$@ … etc).
2. An identifier cannot begin with a digit.
3. A C reserved word cannot be used as an identifier.
4. An identifier defined in a C standard library should not be redefined.

Valid Identifiers

letter_1, letter_2, inches, cent, CENT_PER_INCH, Hello, variable

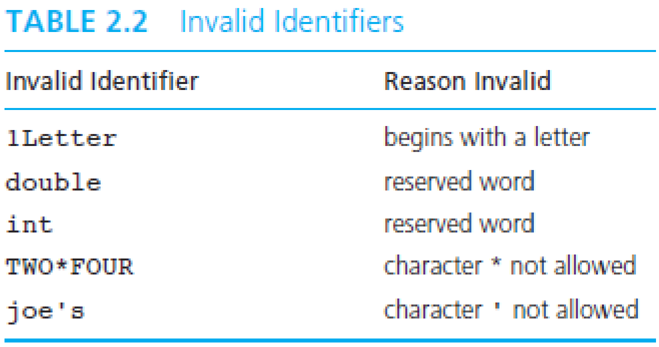


TABLE 2.2 Invalid Identifiers

Invalid Identifier	Reason Invalid
`1Letter`	begins with a letter
`double`	reserved word
`int`	reserved word
`TWO*FOUR`	character * not allowed
`joe's`	character ' not allowed

Initialising Variables

When variables are declared in a program it just means that an appropriate amount of memory is allocated to them for their exclusive use. This memory however is **not initialised** to zero or to any other value automatically and so will contain random values unless specifically initialised before use.

***Syntax* :-** **`type`** **`var-name`** **=** **`constant`** ;

For Example :-

```
char ch = 'a'; //  Character constants are normally represented between single quotes, e.g.
                  'a', 'b', etc
 double d = 12.2323 ;
 int i, j = 20 ; /* note in this case  i is not initialised */
```

There are also a number of special character constants sometimes called *Escape Sequences*, which are preceded by the backslash character '\', and have special meanings in C.

\n	newline
\t	tab
\b	backspace
\'	single quote
\"	double quote
\0	null character

Input/Output Operations and Functions

This section introduces the common input and output functions provided in the C standard library `<stdio>`.

printf()

The printf() function is used for formatted output and uses a control string which is made up of a series of format specifiers to govern how it prints out the values of the variables or constants required. The more common format specifiers are given below

%c	character
%d	signed integer

%i signed integer
%f floating point

For Example :-

```
      int i ;

      printf("%d", i) ;
```

The `printf()` function takes a variable number of arguments. In the above example two arguments are required, the format string and the variable *i*. The value of *i* is substituted for the format specifier %d which simply specifies how the value is to be displayed, in this case as a signed integer.

Some further examples :-

```
int i = 10, j = 20;
char ch = 'a';
double f = 23421.2345;

printf( "%d + %d", i, j ) ;  /* values are substituted from
                      the variable list in order as required */
printf( "%c", ch ) ;

printf( "%s", "Hello World\n" ) ;

printf( "The value of f is : %f", f ) ;/*Output as : 23421.2345 */
```

Field Width Specifiers

Field width specifiers are used in the control string to format the numbers or characters output appropriately .

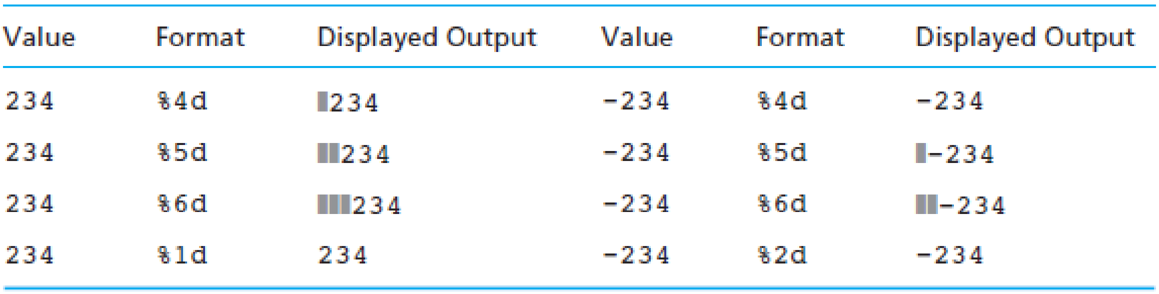
Syntax :- `%[total width printed][.decimal places printed]format specifier]`

where square braces indicate optional arguments.

For Example :-

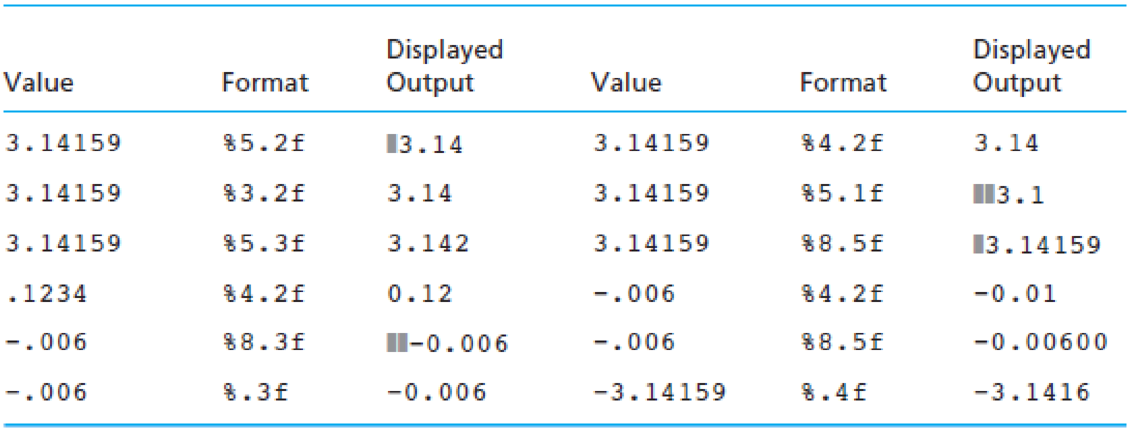
```
      int i = 15, j = -13 ;
      float f = 13.3576 ;
      printf( "%5d\n", i ) ;   /* prints "___15" where _ indicates a space
                               character … here you have 3 spaces!! */
      printf( "%-5d\n", i ) ; /*prints 15___ where 15 is left justified */
      printf( "%05d\n", i ) ; /*prints 00015 □ 0 (zero) causes a field to
                               be padded using zeros rather than space
                               characters */
      printf( "%+d\n", j ) ; /*prints: -13 □ + (plus sign) displays a plus
                               sign preceding positive values and a minus
                               preceding negative values, */

      printf( "%6.2f\n", f ) ; /* prints "_13.36" which has a total width
                               of 6 and displays 2 decimal places  */
      printf( "%*.*f\n", 6,2,f ) ;   /* prints  "_13.36" as above. Here *
                               is used as replacement character for field
                               widths     */
```

TABLE 2.14 Displaying 234 and –234 Using Different Placeholders



Value	Format	Displayed Output	Value	Format	Displayed Output
234	%4d	▮234	-234	%4d	-234
234	%5d	▮▮234	-234	%5d	▮-234
234	%6d	▮▮▮234	-234	%6d	▮▮-234
234	%1d	234	-234	%2d	-234

TABLE 2.16 Formatting Type double Values



Value	Format	Displayed Output	Value	Format	Displayed Output
3.14159	%5.2f	▮3.14	3.14159	%4.2f	3.14
3.14159	%3.2f	3.14	3.14159	%5.1f	▮▮3.1
3.14159	%5.3f	3.142	3.14159	%8.5f	▮3.14159
.1234	%4.2f	0.12	-.006	%4.2f	-0.01
-.006	%8.3f	▮▮-0.006	-.006	%8.5f	-0.00600
-.006	%.3f	-0.006	-3.14159	%.4f	-3.1416

scanf()

This function is similar to the printf function except that it is used for formatted input. The format specifiers have the same meaning as for `printf()` and the space character or the newline character are normally used as delimiters between different inputs.

For Example :-

```
int i, d ;
char c ;
float f ;

scanf( "%d", &i ) ;

scanf( "%d %c %f", &d, &c, &f ) ; /* e.g. type "10_x_1.234" */

scanf( "%d %c", &i, &c ) ;             /* e.g.  type "10 x"  */
```

The & character is the *address of* an operator in C, it returns the address in memory of the variable it acts on. (The & character tells the function exactly where the variable resides in memory and so allow the function to alter it directly rather than to uselessly alter a copy of it.)

```
scanf("%c%c%c", &letter_1, &letter_2, &letter_3);
```

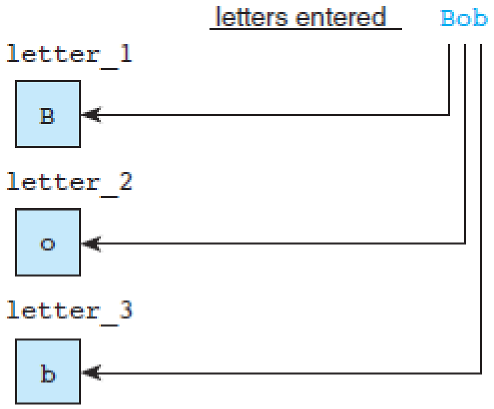


TABLE 2.8 Placeholders in Format Strings

Placeholder	Variable Type	Function Use
`%c`	`char`	`printf/scanf`
`%d`	`int`	`printf/scanf`
`%f`	`double`	`printf`
`%lf`	`double`	`scanf`

Operators

One of the most important features of C is that it has a very rich set of built in operators including arithmetic, relational, logical, and bitwise operators.

Assignment Operator

```
int x ;
x = 20 ;
```

Some common notation :-   lvalue -- left hand side of an assignment operation
rvalue -- right hand side of an assignment operation

Type Conversions :- the value of the rvalue of an assignment is converted to the type of the lvalue. This may sometimes yield compiler warnings if information is lost in the conversion.

For Example :-

```
int x ;
char ch ;
float f ;
ch = x ;   /* ch is assigned lower 8 bits of x, the remaining
           bits are discarded so we have a possible
           information loss  */
x = f ;    /* x is assigned non fractional part of f only
           within int range, information loss possible  */
f = x ;    /* value of x is converted to floating point */
```

Multiple assignments are possible to any degree in C, the assignment operator has right to left associativity which means that the rightmost expression is evaluated first.

For Example :-

```
        x = y = z = 100 ;
```

In this case the expression `z = 100` is carried out first. This causes the value 100 to be placed in z with the value of the whole expression being 100 also. This expression value is then taken and assigned by the next assignment operator on the left i.e. `x = y = ( z = 100 );`

Arithmetic Operators

+ - * / same rules as mathematics with * and / being evaluated before + and -.
% modulus / remainder operator

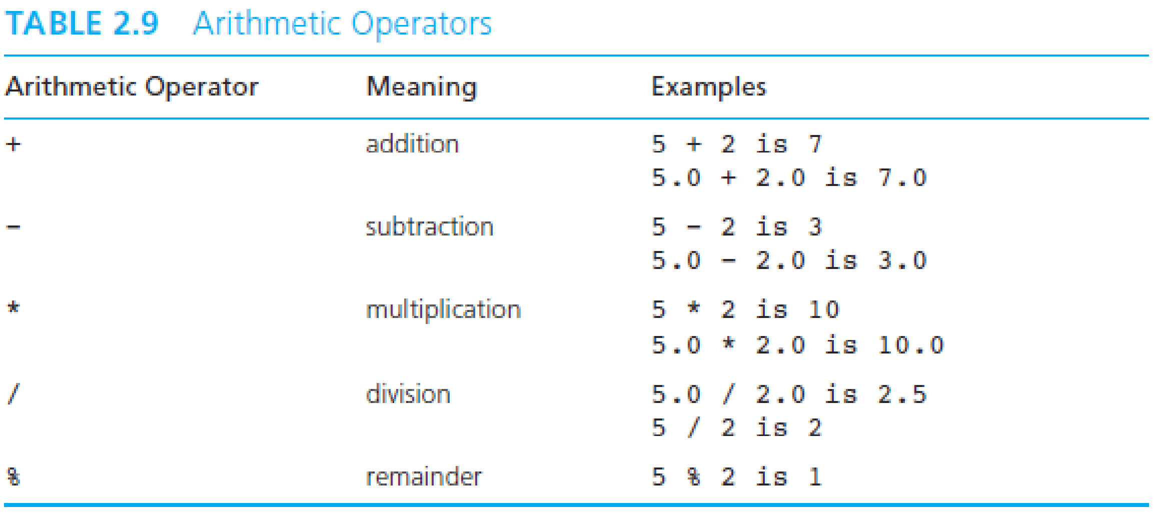


TABLE 2.9 Arithmetic Operators

Arithmetic Operator	Meaning	Examples

+	addition	`5 + 2 is 7`
		`5.0 + 2.0 is 7.0`

–	subtraction	`5 – 2 is 3`
		`5.0 – 2.0 is 3.0`

*	multiplication	`5 * 2 is 10`
		`5.0 * 2.0 is 10.0`

/	division	`5.0 / 2.0 is 2.5`
		`5 / 2 is 2`

%	remainder	`5 % 2 is 1`

For Example :-

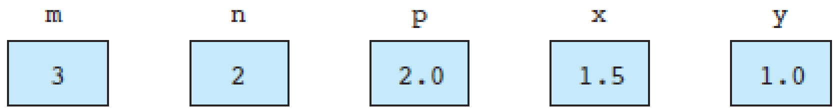
```
int a = 5, b = 2, x ;
float c = 5.0, d = 2.0, f ;

x = a / b ; //  integer division, x = 2.
f = c / d  ;      //  floating point division, f = 2.5.
x = 5 % 2 ; //  remainder operator, x = 1.

x = 7 + 3 * 6 / 2 - 1 ;// x=15,* and / evaluated ahead of + and -.
```

Note that parentheses may be used to clarify or modify the evaluation of expressions of any type in C in the same way as in normal arithmetic.

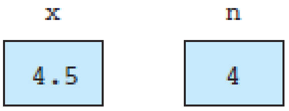
```
x = 7 + ( 3 * 6 / 2 ) - 1 ;// clarifies order of evaluation without penalty
x = ( 7 + 3 ) * 6 / ( 2 - 1 ) ;// changes order of evaluation, x = 60 now.
```

Mixed-Type Assignment Statement

```
      m = 3;
      n = 2;
      p = 2.0;
      x = m / p; /*int/double results double */
      y = m / n; /*int/int  reslts an int number, since y is a double
      variable the result will be saved as a double format 1.0*/
```



```
x = 9 * 0.5; // x is a double variable
n = 9 * 0.5; // n is an it variable
```



Type Conversion through Casts

C allows the programmer to convert the type of an expression by placing the desired type in parentheses before the expression, an operation called a ***type cast.***

Syntax : **`( type ) expression`**

For example, if we have an integer x, and we wish to use floating point division in the expression x/2 we might do the following

```
( float ) x  /  2
```

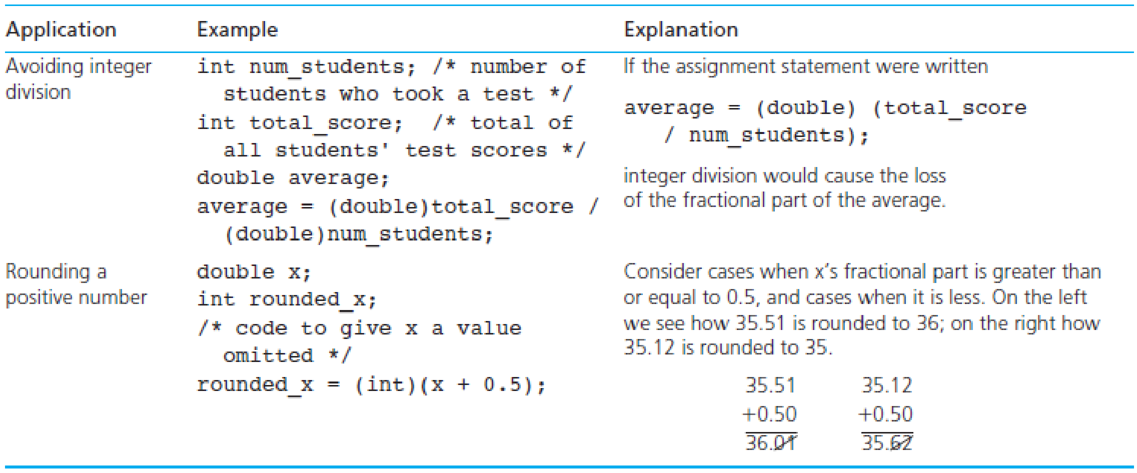
which causes x to be temporarily cast to a floating point value and then implicit casting causes the whole operation to be floating point division.

The same results could be achieved by stating the operation as

```
x  /  2.0
```

which essentially does the same thing but the former is more obvious and descriptive of what is happening.

TABLE 2.12 Examples of the Use of Type Casts



Application	Example	Explanation

Avoiding integer division	`int num_students; /* number of students who took a test */`
	`int total_score; /* total of all students' test scores */`
	`double average;`
	`average = (double)total_score / (double)num_students;`
		If the assignment statement were written
		`average = (double) (total_score / num_students);`
		integer division would cause the loss of the fractional part of the average.

Rounding a positive number	`double x;`
	`int rounded_x;`
	`/* code to give x a value omitted */`
	`rounded_x = (int)(x + 0.5);`
		Consider cases when x's fractional part is greater than or equal to 0.5, and cases when it is less. On the left we see how 35.51 is rounded to 36; on the right how 35.12 is rounded to 35.
		35.51 35.12
		+0.50 +0.50
		36.~~01~~ 35.~~62~~

Increment and Decrement Operators

There are two special unary operators in C, Increment ++, and Decrement --, which cause the variable they act on to be incremented or decremented by 1 respectively.

For Example :-

```
      x++ ;      /* equivalent to      x = x + 1 ;    */
```

++ and -- can be used in prefix or postfix notation. In prefix notation the value of the variable is either incremented or decremented and is then read while in postfix notation the value of the variable is read first and is then incremented or decremented.

For Example :-

```
int i,  j = 2 ;

i = ++ j  ;      /* prefix  :-   i has value 3, j has value 3  */
i = j++ ;  /* postfix  :-  i  has value 3, j has value 4   */
```

Special Assignment Operators

Many C operators can be combined with the assignment operator as shorthand notation

For Example :-

```
      x = x + 10 ;
```

can be replaced by

```
      x += 10 ;
```

Similarly for `-=,  *=,   /=,  %=`, etc.

These shorthand operators improve the speed of execution as they require the expression, the variable x in the above example, to be evaluated once rather than twice.

Relational Operators

The full set of relational operators are provided in shorthand notation

```
 >    >=    <     <=    ==    !=
```

For Example :-

```
      if ( x == 2 )
            printf( "x is equal to 2\n" ) ;
```

Logical Operators

&&	--	Logical AND
||	--	Logical OR
!	--	Logical NOT

For Example :-

```
if (  x >= 0 && x < 10  )
  printf( "x is greater than or equal to zero and less than
          ten.\n" ) ;
```

NB : There is no Boolean type in C so TRUE and FALSE are deemed to have the following meanings.

FALSE -- value zero
TRUE -- any non-zero value but 1 in the case of in-built relational operations

For Example :-

2 > 1	-- TRUE so expression has value 1
2 > 3	-- FALSE so expression has value 0
i = 2 > 1 ;	-- relation is TRUE -- has value 1, i is assigned value 1

NB : Every C expression has a value. Typically, we regard expressions like 2 + 3 as the only expressions with actual numeric values. However, the relation 2 > 1 is an expression which evaluates to TRUE so it has a value 1 in C. Likewise if we have an expression x = 10 this has a value which in this case is 10 the value actually assigned.

Sizeof Operator

The sizeof operator gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types as we will see later on).

The expression is either an identifier or a type-cast expression (a type specifier enclosed in parentheses).

Syntax : ***`sizeof ( expression )`***

For Example :-

```
int x , size ;

size = sizeof ( x ) ;
printf("The integer x requires %d bytes on this machine", size);

printf( "Doubles take up %d bytes on this machine", sizeof (
double)) ;
```

Precedence of Operators

When several operations are combined into one C expression the compiler has to rely on a strict set of precedence rules to decide which operation will take preference. The precedence of C operators is given below.

Precedence	**Operator**	**Associativity**
Highest	( ) [ ] → .	left to right
	! ++ -- +(unary) -(unary) (type) * & sizeof	right to left
	* / %	left to right
	+ -	left to right
	< <= > >=	left to right
	== !=	left to right
	&&	left to right
	||	left to right
	= += -= *= /= %=	right to left

Operators at the top of the table have highest precedence and when combined with other operators at the same expression level will be evaluated first.

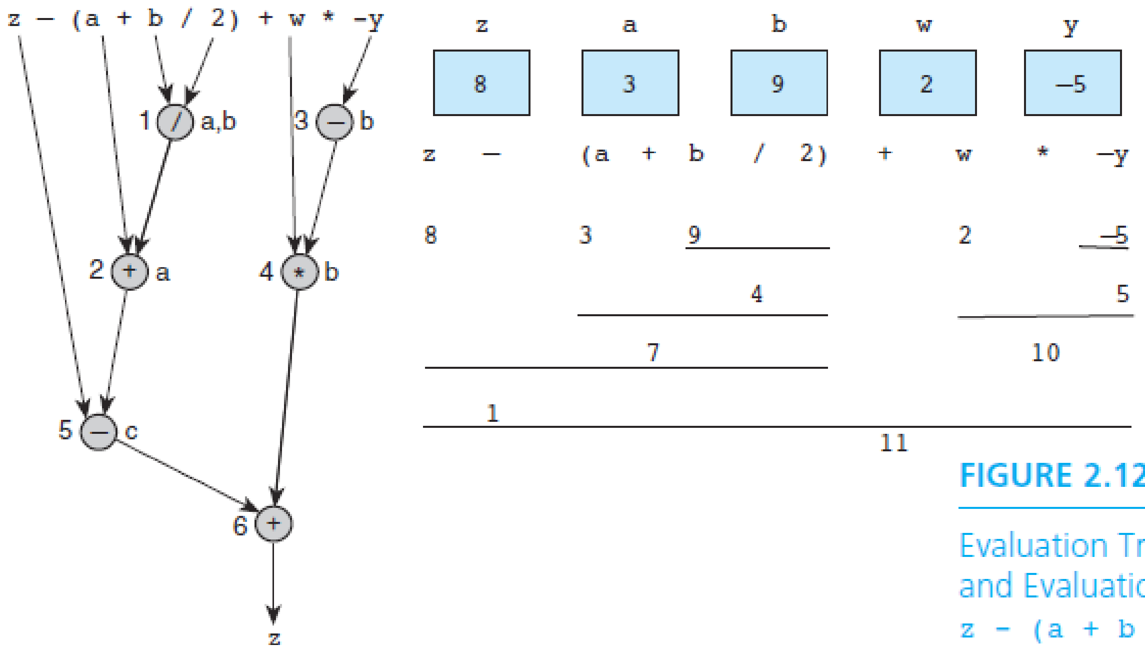


FIGURE 2.12 Evaluation Tree and Evaluation for `z - (a + b / 2) + w * -y`

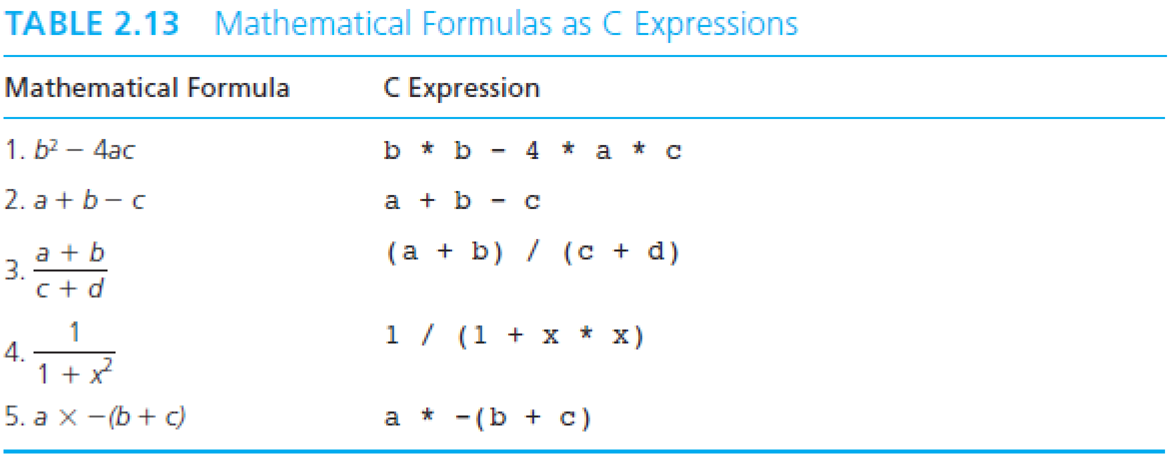


TABLE 2.13 Mathematical Formulas as C Expressions

Mathematical Formula	C Expression
1. $b^2 - 4ac$	`b * b - 4 * a * c`
2. $a + b - c$	`a + b - c`
3. $\frac{a + b}{c + d}$	`(a + b) / (c + d)`
4. $\frac{1}{1 + x^2}$	`1 / (1 + x * x)`
5. $a \times -(b + c)$	`a * -(b + c)`

File I/O

Input and output to and from files is identical to that at the command line, except the fprintf and fscanf functions are used and they require another argument. This additional argument is called a `file pointer`.

In order to write two floating point numbers to a file

1. First you need to declare the file pointer with the FILE type
2. open the file
3. use the `fprintf` function to write in the file

```
float x=1, y=2;
FILE *file;
file = fopen(‘‘file.txt’’,’’w’’);
fprintf(file,’’%f %f\n’’,x,y);
fclose(file);
```

The function *fprintf* is identical to the *printf* function, except now we see it has another argument: “file”, which is a pointer to the file. Before you use the file variable, you need to open the file with

```
file = fopen(“file.txt”,”w”);
```

This opens up the file ‘‘file.txt’’ and the ‘‘w’’ which is the mode and indicates how the file will be used. The following three modes are allowed:

Mode	String
Open for reading	“r”
Open for writing “	w”

When you are done with the file, you close it with

```
fclose(file);
```

The ‘‘r’’ mode is used when you would like to open a file for reading. To read two floating point numbers from a file, you would use the fscanf function, which is identical to scanf, except that it takes the file pointer as its first argument, as in

```
float x, y;
FILE *file;
file = fopen(‘‘file.txt’’,’’r’’);
fscanf(file,’’%f %f\n’’,&x,&y);
fclose(file);
```

You can check to make sure your files are opened correctly (that is, that they exist), by checking to make sure file is not the predefined NULL pointer. We’ll discuss this pointer in more detail later, but for now, to ensure that your file was opened correctly, you use

```
if(!file)
     printf(‘‘File did not open correctly!\n’’);
```

Exercises

1. Write a program to check what the following code segment outputs and explain the results.

```
char c ;
printf("sizeof( c ) = %d\n", sizeof( c ) ) ;
printf("sizeof( 'a' ) = %d\n", sizeof( 'a' ) ) ;
printf("sizeof( c = 'a' ) = %d\n", sizeof( c='a' ) ) ;
```

2. Write a program which reads a character from the keyboard and writes out its ASCII representation.
   Now write a program which reads in an integer from the keyboard and print out its character representation. Make certain you carry out appropriate bounds / error checking.

3. Describe the output from each of the following statements.

```
i.  printf( “%-10d\n”, 10000 ) ;
ii. printf( “%8.3f\n”, 23.234 ) ;
iii.printf( “%+*.*lf\n”, 10, 3, 1234.234 ) ;
iv. printf( “%d\n”, 16 ) ;
```

4. What value does x contain after each of the following where **x is of type float**.

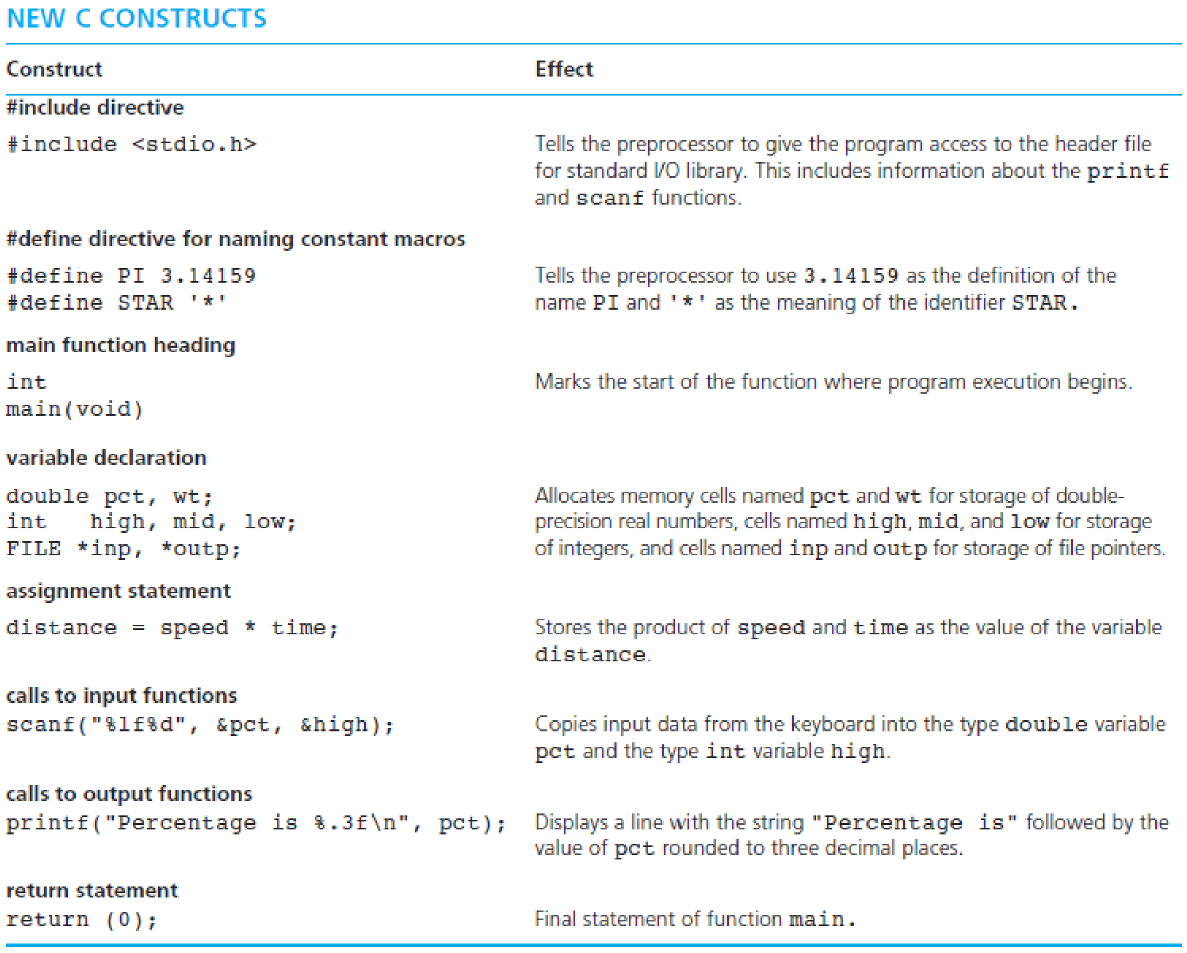
```
i.    x = 7 + 3 * 6 / 2 - 1 ;
```

```
ii. x = 2 % 2 + 2 * 2 - 2 / 2 ;
iii.x = ( 3  * 9 * ( 3 + ( 4 * 5 / 3 ) ) ) ;
iv. x = 12.0 + 2 / 5 * 10.0 ;
v.  x = 2 / 5 + 10.0 * 3 - 2.5 ;
vi. x = 15 > 10 && 5 < 2 ;
```

5. Write a program to read Fahrenheit temperatures and print them in Celsius. The formula is

6. °C = (5/9)(°F - 32). Use variables of type double in your program.

7. Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Use the value 3.14159 for "pi".



NEW C CONSTRUCTS

Construct	Effect
#include directive	
`#include <stdio.h>`	Tells the preprocessor to give the program access to the header file for standard I/O library. This includes information about the `printf` and `scanf` functions.
#define directive for naming constant macros	

`#define PI 3.14159`
`#define STAR '*'`
	Tells the preprocessor to use `3.14159` as the definition of the name `PI` and `'*'` as the meaning of the identifier `STAR`.

main function heading	

`int`
`main(void)`
	Marks the start of the function where program execution begins.

variable declaration	

`double pct, wt;`
`int    high, mid, low;`
`FILE *inp, *outp;`
	Allocates memory cells named `pct` and `wt` for storage of double-precision real numbers, cells named `high`, `mid`, and `low` for storage of integers, and cells named `inp` and `outp` for storage of file pointers.

assignment statement	
`distance = speed * time;`	Stores the product of `speed` and `time` as the value of the variable `distance`.
calls to input functions	
`scanf("%lf%d", &pct, &high);`	Copies input data from the keyboard into the type `double` variable `pct` and the type `int` variable `high`.
calls to output functions	
`printf("Percentage is %.3f\n", pct);`	Displays a line with the string `"Percentage is"` followed by the value of `pct` rounded to three decimal places.
return statement	
`return (0);`	Final statement of function `main`.

Chapter 2: Top-Down Design with Functions

In programming, a function is a segment that groups a set of code statements in a given order and that can be referenced by a unique name to perform a specific task.

A C program has at least one function main(). Without main() function, there is technically no C program.

Types of C functions

There are two types of functions in C programming:

- Library function
- User defined function

Library functions

C promotes reuse by providing many predefined functions that can be used to perform mathematical computations.

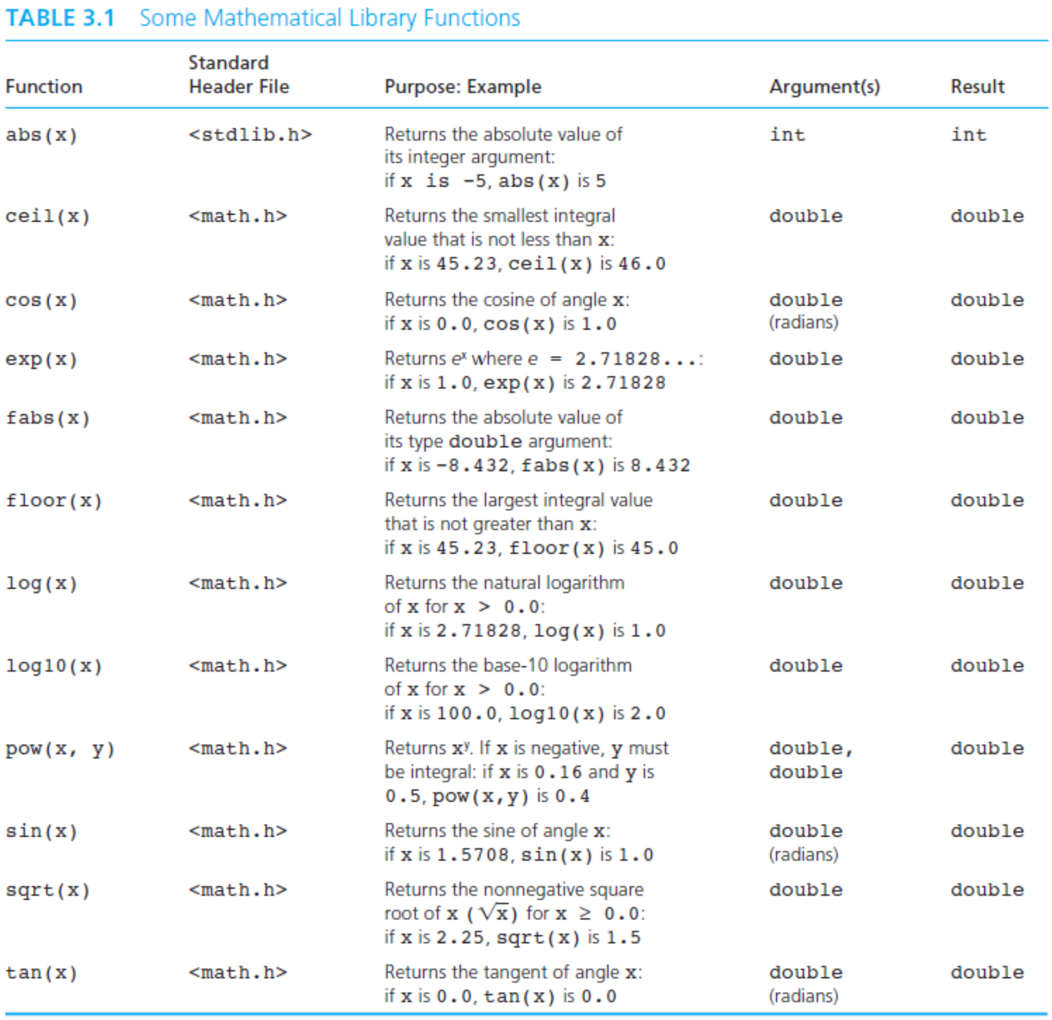


TABLE 3.1 Some Mathematical Library Functions

Function	Standard Header File	Purpose: Example	Argument(s)	Result
`abs(x)`	`<stdlib.h>`	Returns the absolute value of its integer argument: if `x is -5`, `abs(x)` is `5`	`int`	`int`
`ceil(x)`	`<math.h>`	Returns the smallest integral value that is not less than `x`: if `x` is `45.23`, `ceil(x)` is `46.0`	`double`	`double`
`cos(x)`	`<math.h>`	Returns the cosine of angle `x`: if `x` is `0.0`, `cos(x)` is `1.0`	`double` (radians)	`double`
`exp(x)`	`<math.h>`	Returns e^x where e `= 2.71828...`: if `x` is `1.0`, `exp(x)` is `2.71828`	`double`	`double`
`fabs(x)`	`<math.h>`	Returns the absolute value of its type `double` argument: if `x` is `-8.432`, `fabs(x)` is `8.432`	`double`	`double`
`floor(x)`	`<math.h>`	Returns the largest integral value that is not greater than `x`: if `x` is `45.23`, `floor(x)` is `45.0`	`double`	`double`
`log(x)`	`<math.h>`	Returns the natural logarithm of `x` for `x > 0.0`: if `x` is `2.71828`, `log(x)` is `1.0`	`double`	`double`
`log10(x)`	`<math.h>`	Returns the base-10 logarithm of `x` for `x > 0.0`: if `x` is `100.0`, `log10(x)` is `2.0`	`double`	`double`

`pow(x, y)`	`<math.h>`	Returns x^y. If `x` is negative, `y` must be integral: if `x` is `0.16` and `y` is `0.5`, `pow(x,y)` is `0.4`	`double,`
			`double`
				`double`

`sin(x)`	`<math.h>`	Returns the sine of angle `x`: if `x` is `1.5708`, `sin(x)` is `1.0`	`double` (radians)	`double`
`sqrt(x)`	`<math.h>`	Returns the nonnegative square root of `x` ($\sqrt{x}$) for `x` ≥ `0.0`: if `x` is `2.25`, `sqrt(x)` is `1.5`	`double`	`double`
`tan(x)`	`<math.h>`	Returns the tangent of angle `x`: if `x` is `0.0`, `tan(x)` is `0.0`	`double` (radians)	`double`

Examples of using C Library functions:

1. Square root function

```
#include <stdio.h>
#include <math.h>
int main(){
   float num,root;
   printf("Enter a number to find square root.");
   scanf("%f",&num);
   root=sqrt(num);/* Computes the square root of num and stores
in root. */
   printf("Square root of %.2f=%.2f",num,root);
   return 0;
}
```

Output

```
Enter a number to find square root.12
Square root of 12.00=3.46
```

2. Power function

```
#include <stdio.h>
#include <math.h>

int main ()
{
   printf("Value 8.0 ^ 3 = %lf\n", pow(8.0, 3));
   printf("Value 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));
   return(0);
}
```

Output:

```
Value 8.0 ^ 3 = 512.000000
Value 3.05 ^ 1.98 = 9.097324
```

3. More Examples

```
#include <stdio.h>
#include <math.h>
int main()
{
   float result,value;
   printf("Input a float value: ");
   scanf("%f",&value);
   result = sqrt(value);
   printf("The square root of %.2f is %.2f\n", value,result);
   result = pow(value,3);
   printf("%.2f to the 3rd power is %.2f\n", value,result);
   result = floor(value);
   printf("The floor of %.2f is %.2f\n", value,result);
   result = ceil(value);
   printf("And the ceiling of %.2f is %.2f\n", value,result);
   return(0);
}
```

Output:

```
Input a float value: 9.3
The square root of 9.30 is 3.05
9.30 to the 3rd power is 804.36
The floor of 9.30 is 9.00
And the ceiling of 9.30 is 10.00
```

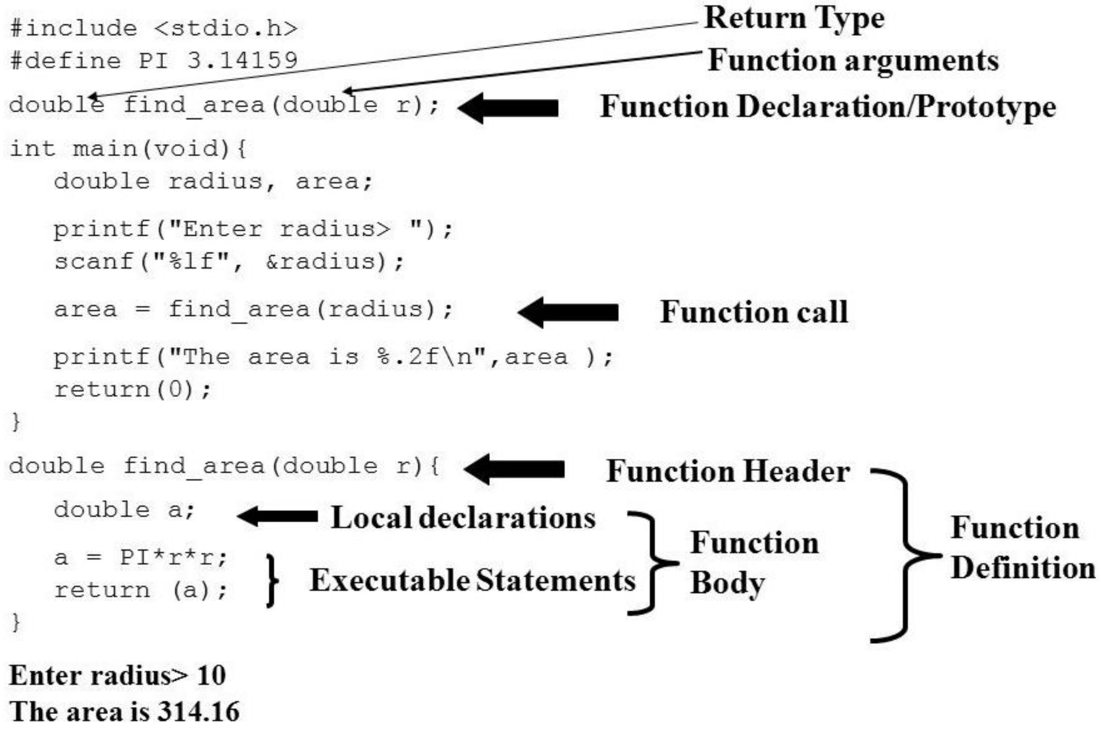
User defined functions

C allows programmer to define their own function according to their requirement. These types of functions are known as user-defined functions. Suppose, a programmer wants to find factorial of a number and check whether it is prime or not in same program. Then, he/she can create two separate user-defined functions in that program: one for finding factorial and other for checking whether it is prime or not.

Syntax :
```
return_type function_name (  parameter_list )
{
      body of function ;
}
```

Example of user-defined function

Write a C program to find the area of a circle. Make a function to find the area and display the result in main() function.



```
#include <stdio.h>
#define PI 3.14159
double find_area(double r);
int main(void){
   double radius, area;
   printf("Enter radius> ");
   scanf("%lf", &radius);
   area = find_area(radius);
   printf("The area is %.2f\n",area );
   return(0);
}
double find_area(double r){
   double a;
   a = PI*r*r;
   return (a);
}
```

```
Enter radius> 10
The area is 314.16
```

Function Prototype (declaration)

When writing programs in C it is normal practice to write the main() function first and to position all user functions after it or indeed in another file. Thus if a user function is called directly in main() the compiler will not know anything about it at this point i.e. if it takes parameters etc. This means we need to give the compiler this information by providing a function prototype or declaration before the function is called.

Syntax : `return_type function_name (  parameter_list );`

This declaration simply informs the compiler what type the function returns and what type and how many parameters it takes. Names may or may not be given to the parameters at this time.

For Example :- “Hello World” program.

```
#include <stdio.h>       /* standard I/O function prototypes */
void hello(  ) ;         /* prototype  */
void main( void )
{
  hello () ;                   //  function call
}
void hello ( )                 // function definition
{
  printf ( "Hello World \n" ) ;
}
```

Function Definition

The prototype tells the compiler what arguments the function takes and what it returns, but not what it does. We define our own functions just like we do the main function

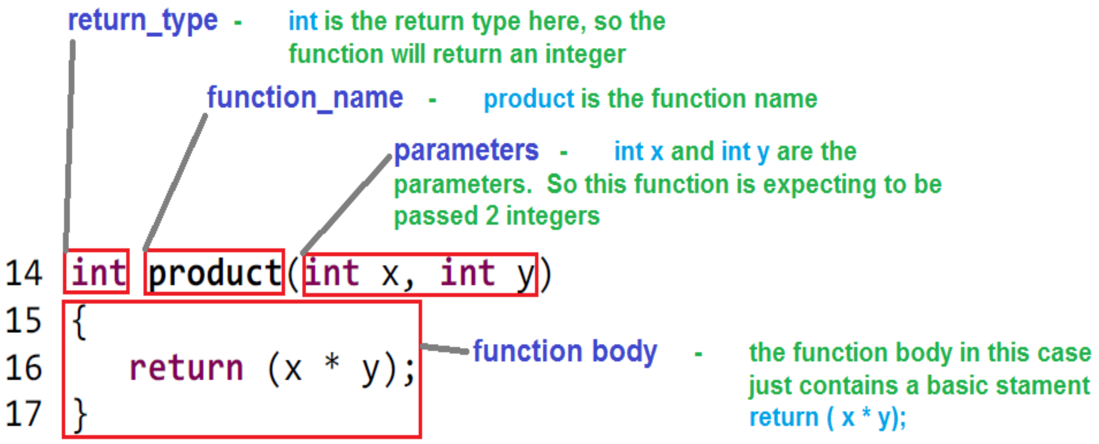
- Function Header – The same as the prototype, except it is not ended by the symbol ;
- Function Body – A code block enclosed by {}, containing variable declarations and executable statements. In the function body, we define what actually the function does.

Function Call

To use a function, you will have to call that function to perform the defined task.
When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

- To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

One more Example:



```
int product(int x, int y)
{
    return (x * y);
}
```

int is the return type here, so the function will return an integer

product is the function name

int x and int y are the parameters. So this function is expecting to be passed 2 integers

the function body in this case just contains a basic stament return (x * y);

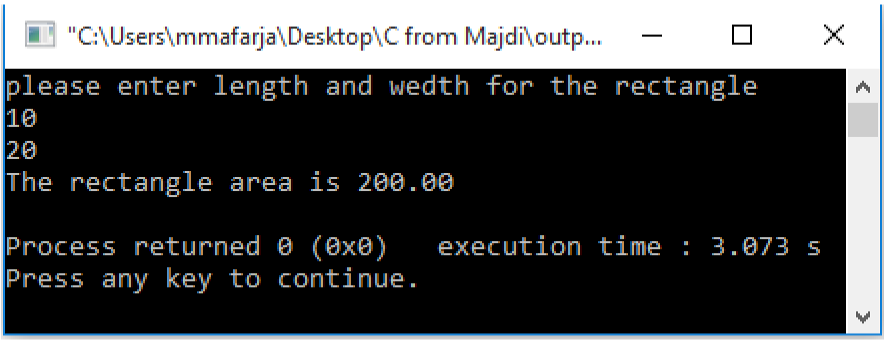
Types of User-defined Functions

For better understanding of arguments and return type in functions, user-defined functions can be categorized as:

1. Function with no arguments and no return value
2. Function with no arguments and return value
3. Function with arguments but no return value
4. Function with arguments and return value.

The following program aims to find the rectangle area. I'll use this example to explain all types of functions.

```
#include<stdio.h>
int main()
{
    double lenght, width, area;
    printf("please enter length and wedth for the rectangle\n");
    scanf("%lf%lf",&lenght,&width);
    area = lenght*width;
    printf("The rectangle area is %f\n",area);
    return 0;
}
```



```
please enter length and wedth for the rectangle
10
20
The rectangle area is 200.00

Process returned 0 (0x0)   execution time : 3.073 s
Press any key to continue.
```

1. Function with no arguments and no return value

```
void find_Area (  ) // Function Header
{ // open brace
    double lenght, width, area;
    printf("please enter length and wedth for the rectangle\n");
    scanf("%lf%lf",&lenght,&width);
    area = lenght*width;
    printf("The rectangle area is %f\n",area);
} // close brace
```

How to use function 1?

You have to declare the function prototype at the beginning of the program (same return data type, function name and number of parameters, then add semicolon(;)).

```
void find_Area(); // Function prototype
```

In the main function; since the function has no return data, so just put the function name and () to call the function.

```
find_Area(); // Function Call
```

See the following code …

```
#include<stdio.h>
void find_Area(); // Function prototype
int main()
{
    find_Area(); // Function Call
    return 0;
}
/*function definition */
void find_Area() // Function Header
{
    double lenght, width, area;
    printf("please enter length and wedth for the rectangle\n");
    scanf("%lf%lf",&lenght,&width);
    area = lenght*width;
    printf("The rectangle area is %f\n",area);
}
```

2. Function with no arguments and return value

```
#include<stdio.h>
double find_Area();
int main()
{
    double a = find_Area(); // the function returns the area and
  save it to the a variable
    printf("The rectangle area is %f\n", a);
    return 0;
}
double find_Area()
{
    double lenght, width, area;
    printf("please enter length and wedth for the rectangle\n");
    scanf("%lf%lf",&lenght,&width);
    area = lenght*width;
    return area;
}
```

How to use function 2?

1. Don’t forget to add the function prototype.
2. Declare a variable with same type of the returned data (i.e., the function returns area, area is a double, so you have to declare ‘a’ as double…).
3. Assign the value returned by the function to the declared variable (a).
   `double a = find_Area();`

OR you can call it directly in the printf function if you don’t need to use the returned value from the function.

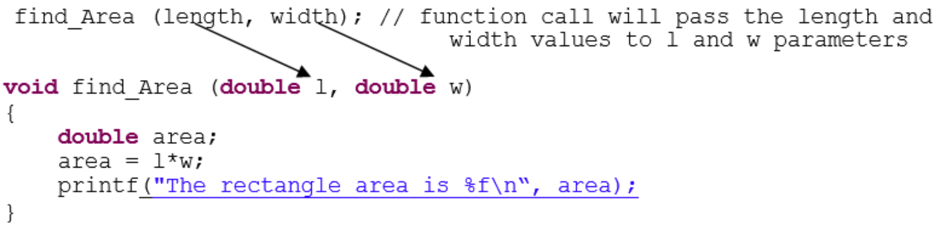
```
printf("The rectangle area is %f\n", find_Area());
```

3.Function with arguments but no return value

```
#include<stdio.h>
void find_Area(double l, double w);
int main()
{
    double length, width;
    printf("please enter length and wedth for the rectangle\n");
    scanf("%lf%lf",&length,&width);
    find_Area(length,width);
    return 0;
}
void find_Area(double l, double w)
{
     double area;
     area = l*w;
     printf("The rectangle area is %f\n‟, area);
}
```

How to use function 3?

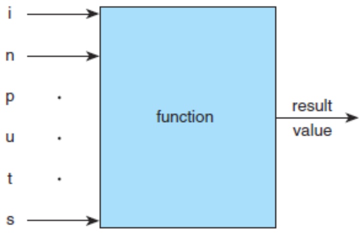
1. After declaring the function prototype before the main() function…
2. Write the function name in the main function and **Pass** values to the function be assigned to each parameter.



```
find_Area (length, width); // function call will pass the length and
                               width values to l and w parameters

void find_Area (double l, double w)
{
    double area;
    area = l*w;
    printf("The rectangle area is %f\n‟, area);
}
```

4. Function with arguments and return value.

```
#include<stdio.h>
double find_Area(double l, double w);
int main()
{
     double length, width;
     printf("please enter length and wedth for the rectangle\n");
     scanf("%lf%lf",&length,&width);
     double a = find_Area(length,width);
     printf("The rectangle area is %f\n",a)
     return 0;
}
double find_Area(double l, double w)
{
     double area;
     area = l*w;
     return area;
}
```



How to use function 3?

1. After declaring the function prototype before the main() function…
2. Declare a variable with same type of the returned data; double in this example.
3. Assign the value returned by the function to the declared variable (a).
4. Pass values to the function be assigned to each parameter.
5. In this function we have two parameters, so we have to pass two double values, the values may be constants (1, 2.3, 4 etc.) or can be passed by variables (x, i).

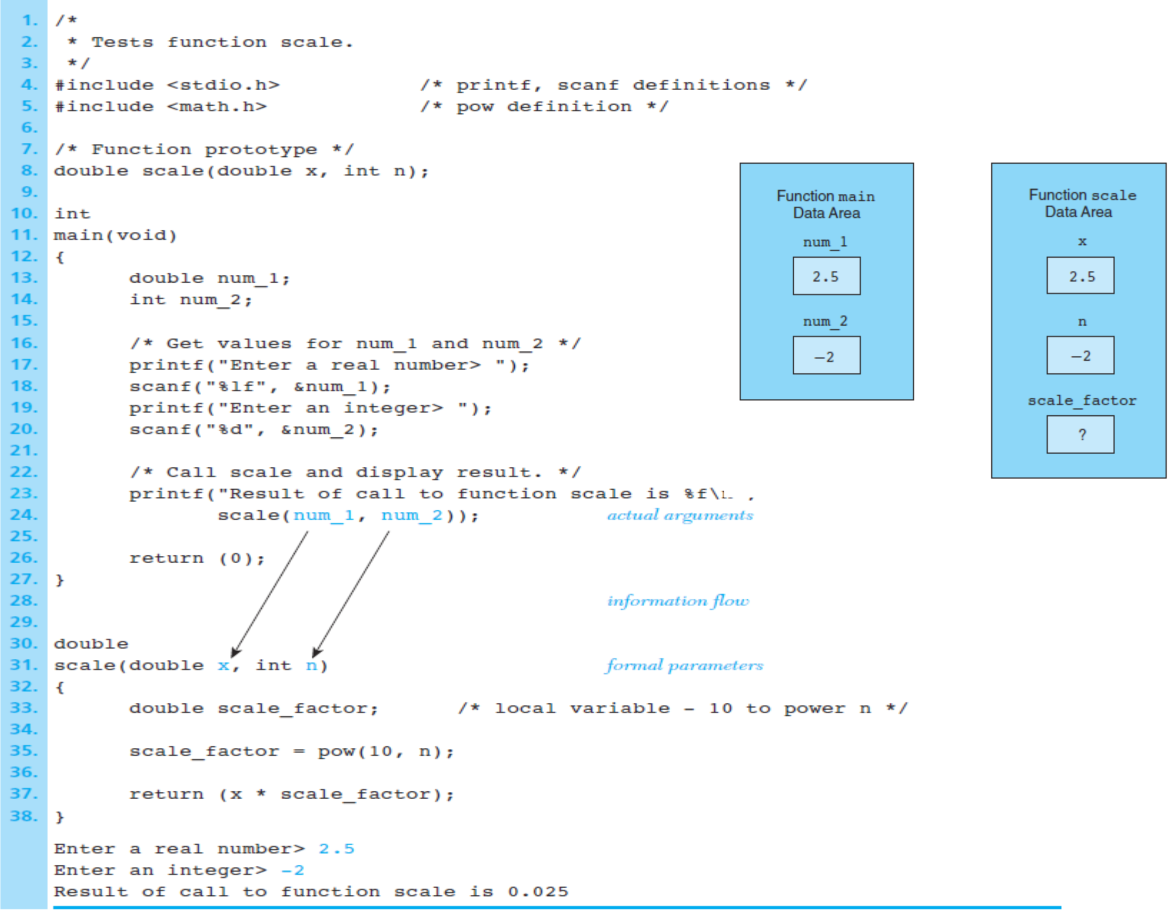
Example:

Write a complete C program that finds the maximum between two numbers. Use function to find the max.

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
   /* calling a function to get max value */
   ret = max(a, b);
   printf( "Max value is : %d\n", ret );
   return 0;
}
/* function returning the max between two numbers */
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;
}
```

Another example to show you what's happened to the computer memory when using functions:

FIGURE 3.24 Testing Function scale



```
/*
 * Tests function scale.
 */
#include <stdio.h>           /* printf, scanf definitions */
#include <math.h>            /* pow definition */

/* Function prototype */
double scale(double x, int n);

int
main(void)
{
      double num_1;
      int num_2;

      /* Get values for num_1 and num_2 */
      printf("Enter a real number> ");
      scanf("%lf", &num_1);
      printf("Enter an integer> ");
      scanf("%d", &num_2);

      /* Call scale and display result. */
      printf("Result of call to function scale is %f\n",
             scale(num_1, num_2));

      return (0);
}

double
scale(double x, int n)
{
      double scale_factor;      /* local variable - 10 to power n */

      scale_factor = pow(10, n);

      return (x * scale_factor);
}
```

```
Enter a real number> 2.5
Enter an integer> -2
Result of call to function scale is 0.025
```

Exercises

1. Write a program that can convert temperatures from the Fahrenheit scale to Celsius and back. The relationship is °C = (5/9)(°F - 32). Your program should read a temperature and which scale is used and convert it to the other, printing out the results. Write one or more functions to carry out the actual conversion.

2. Write a program that reads in the radius of a circle and prints the circle’s diameter, circumference and area. Write functions for appropriate logical tasks in your program. You should #define all appropriate constants in your program.

Chapter 4: Selection Structures, if and switch Statements

Introduction

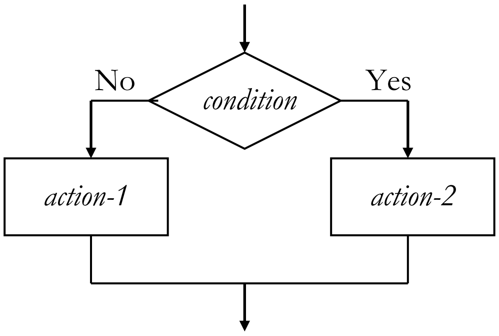
In a procedural language (Like C), the flow of execution is determined by the control structures of the program. All programs can be written using these three control structures:

- Sequence
- Selection (if and switch)
- Repetition (for, while and do-while)

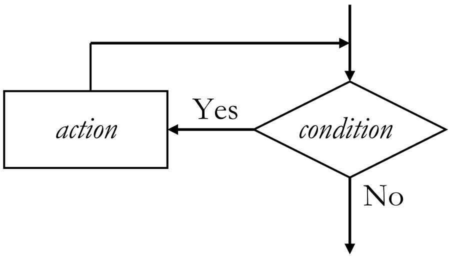
Sequential flow is specified by a compound statement, consisting of a group of statements in a block, delimited by braces:

```
{
        $statement_1$;
        $statement_2$;
                .
                .
        $statement_n$;
}
```

A ***selection*** structure allows for choice among alternative statements, and a ***repetition*** structure allows an action to be repeated while some condition remains true, as depicted in the flowcharts below. We will study these structures in details.



Selection structure



Repetition structure

Selection Structure

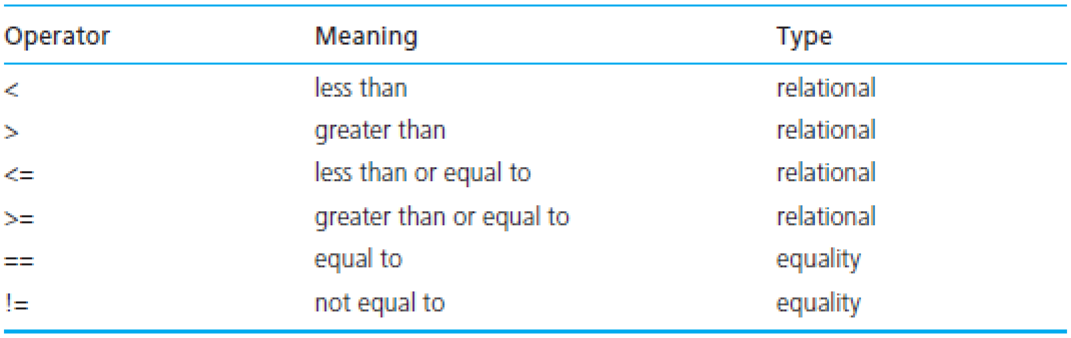
C provides a few versions of the selection structure: one-way branch, two-way branch, and multi-way branch.

Relational and Equality Operators

Most conditions that we use to perform comparisons will have one of these forms:

variable	*relational-operator*	*variable*
variable	*relational-operator*	*constant*
variable	*equality-operator*	*variable*
variable	*equality-operator*	*constant*

TABLE 4.1 Relational and Equality Operators



Operator	Meaning	Type
<	less than	relational
>	greater than	relational
<=	less than or equal to	relational
>=	greater than or equal to	relational
==	equal to	equality
!=	not equal to	equality

Example:

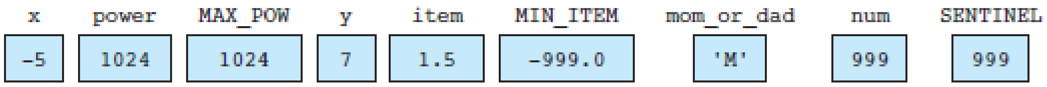
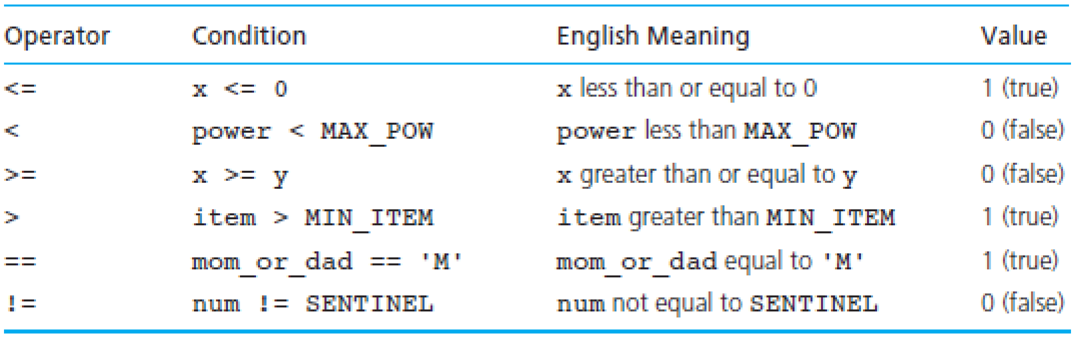


TABLE 4.2 Sample Conditions



Operator	Condition	English Meaning	Value
<=	`x <= 0`	`x` less than or equal to 0	1 (true)
<	`power < MAX_POW`	`power` less than `MAX_POW`	0 (false)
>=	`x >= y`	`x` greater than or equal to `y`	0 (false)
>	`item > MIN_ITEM`	`item` greater than `MIN_ITEM`	1 (true)
==	`mom_or_dad == 'M'`	`mom_or_dad` equal to `'M'`	1 (true)
!=	`num != SENTINEL`	`num` not equal to `SENTINEL`	0 (false)

Logical Operators

&&	--	Logical AND
||	--	Logical OR
!	--	Logical NOT

A	! A	B	A && B	A || B
T	F	T	T	T
T	F	F	F	T
F	T	T	F	T
F	T	F	F	F

Operator Precedence

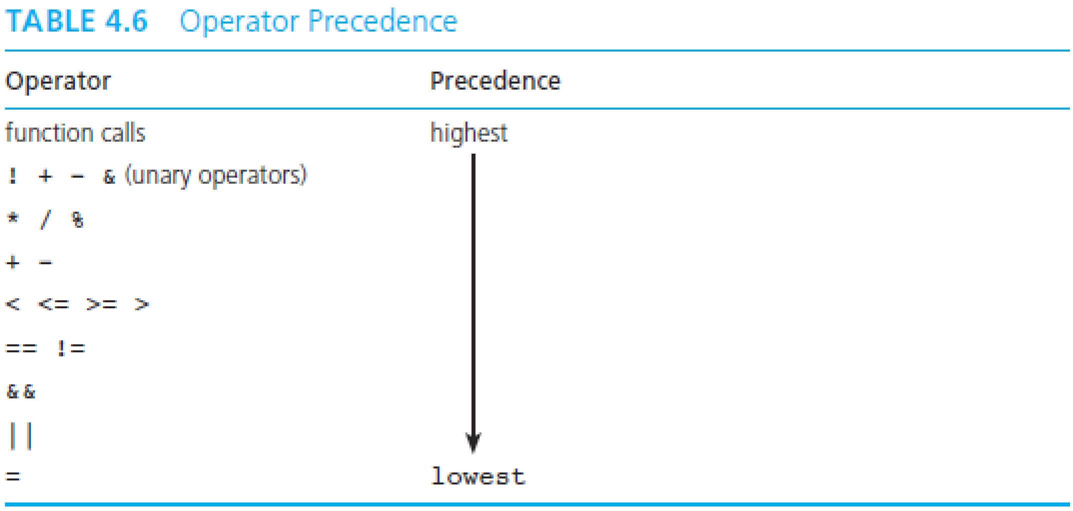
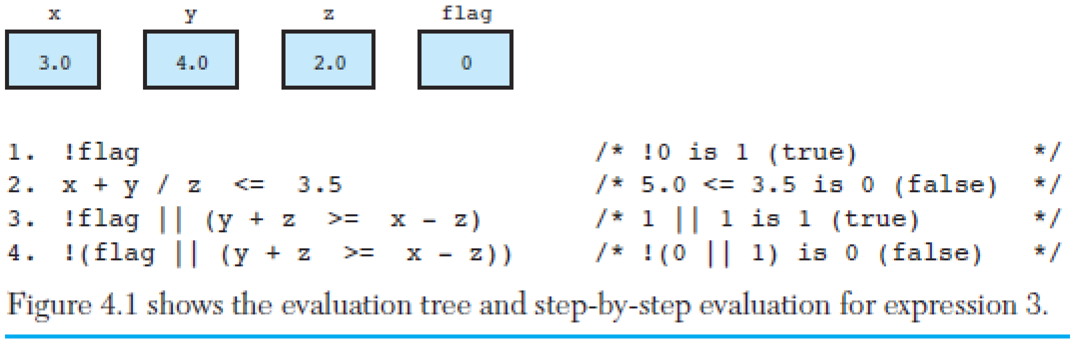


TABLE 4.6 Operator Precedence

Operator	Precedence
function calls	highest
! + - & (unary operators)	
* / %	
+ -	
< <= >= >	
== !=	
&&	
||	
=	lowest

<u>Example</u>


x
3.0
y
4.0
z
2.0
flag
0

```
1.  !flag                                 /* !0 is 1 (true)         */
2.  x + y / z   <=   3.5                  /* 5.0 <= 3.5 is 0 (false) */
3.  !flag || (y + z  >=  x - z)           /* 1 || 1 is 1 (true)      */
4.  !(flag || (y + z  >=  x - z))         /* !(0 || 1) is 0 (false)  */
```

Figure 4.1 shows the evaluation tree and step-by-step evaluation for expression 3.

The 'if' statement

The 'if' statement allows some action to be taken when a condition is met. The action could be a single statement, or a compound statement that comprises a number of statements. The syntax for an 'if' construct with a single-statement action is

if (*expression*)
    statement;

If the *expression* is nonzero (true), then *statement* is executed; otherwise, *statement* is skipped and control passes to the next statement in the program.

For example,

```
fine = 20;
if (speed > 50)
     fine += 10;
printf("Fine is %d\n", fine);
```

Here, ‘fine’ will remain at 20 if the condition (speed > 50) is false. Otherwise, ‘fine’ will be increased to 30.

Compound-statement Action

Sometime, the course of action we want to take when the ‘if’ condition is true consists of a compound statement (a sequence of statements). In this case, we use the braces to mark off the compound statement. The syntax is hence:

```
if (expression)
{
     compound-statement;
}
```

Flags

Making use of the fact that zero is false and non-zero value is true, we could save a few keystrokes in some cases.

For example, suppose ‘attended’ is an integer variable to indicate if a student has attended a course. It contains zero if he has not attended, or 1 if he has. Such a variable that stores zero or 1 is known as a *flag*. Suppose ‘attendance’ is an integer variable that counts the number of students in a course. A code fragment could look like this (this code is most likely within a loop in order to perform the counting):

```
if (attended == 1)
     attendance++;
```

The above could be replaced by:

```
if (attended)
     attendance++;
```

which achieves the same result. Why?

Similarly, if we want to count the number of absentees, we could have:

```
if (attended == 0)
{
   absentees++;
   printf("One more absentee.\n");
}
```

or this shorter version favoured by experienced programmers:

```
if (!attended){
   absentees++;
   printf("One more absentee.\n");

}
```

BE Careful when naming the flag … use a meaningful name …

Nested 'if' statements

Since an 'if' construct is a statement, it can appear in the body of another 'if' construct. This gives rise to nested 'if' statements, as shown below:

```
if (exam >= 80)
   if (project >= 90)
      grade = 'A';
```

The above code could also be written using a single 'if' construct with compound condition:

```
if (exam >= 80 && project >= 90)

   grade = 'A';
```

Lazy (short-circuit) evaluation

The evaluation stops as soon as the value of the logical expression can be determined. The evaluation is carried out from left to right.

In the example above, if the expression (exam >= 80) is false, the second expression (project >= 90) would not need to be evaluated, since the whole compound condition is false.

Similarly, for a compound logical expression like this:

```
if (bank_bal < 0.0 || expenses > 100.0)
   printf("Red alert!\n");
```

If bank_bal is less than 0.0, the whole condition is true, and so the second logical expression need not be evaluated.

Common Mistakes

1. Floating-point numbers are inexact. Hence, you should not use the equality (==) or the inequality (!=) test on floating-point numbers. In testing for equality, compute the difference of the two values instead and check that the difference is small enough. In testing for inequality, you may compute the difference and check that it exceeds a threshold, or you may simply use other relational operators like <, <=, >, >= depending on the situation.

2. The equality (==) operator is often mixed up with the assignment (=) operator, and that could result in hard-to-spot errors. Some beginners also tend to omit the parentheses around the condition.

3. It is a mistake to put a semicolon after the condition, like this:

   ```
   if (a > b);
      printf("a is larger than b\n");
   ```

   This would create an empty statement for the 'if' construct. The printf() statement is now outside the 'if' construct, and thus it will be executed regardless of the truth value of the 'if' condition.

The 'if-else' Construct

The 'if-else' construct provides a two-way branch in which one of the two alternative paths is taken depending on the truth value of the condition. The format of the 'if-else' construct is:

```
     if (expression)
          statement1 ;
     else
          statement2 ;
```

or, with compound-statement actions, we have:

```
     if (expression){
         compound-statement1 ;
     }
     else{
         compound-statement2 ;
     }
```

For example,

```
     if (count == 0)
          ave = 0.0;
     else
          ave = (float) total/count;


     if (score1 < score2){
          better_score = score2;
          printf("score2 is better\n");
     }
     else{
          better_score = score1;
          printf("score1 is better\n");
     }
```

The use of 'if-else' avoids redundant code such as this:

```
       if (count == 0)
          ave = 0.0;
       if (count != 0)
          ave = (float) total/count;
```

In the above code, two conditions will have to be tested. Using the 'if-else' construct, only one condition is necessary.

Logical Assignment for Flags

Consider this example where a flag 'snr_citizen' is assigned 1 to represent a senior citizen, or zero to represent non-senior citizen:

```
       if (age >= 65)
          snr_citizen = 1;
       else
          snr_citizen = 0;
```

In cases like this, we may write a single assignment statement without using the 'if' statement:

```
snr_citizen = (age >= 65);
```

As another example, the code below sets the variable ‘even’ to 1 if n is even, or zero if n is odd.

```
if (n % 2 == 0)
     even = 1;
else
     even = 0;
```

The short form would be:

```
even = (n % 2 == 0);
```

The statement is more succinct, and achieves the same purpose. You are encouraged to use such form whenever appropriate.

Nested ‘if-else’ statements

Just as we have nested ‘if’ statements, we can also have nested ‘if-else’ statements. For example:

```
if (marks < 50)
   grade = 'F';
else
   if (marks < 70)
      grade = 'B';
   else
      grade = 'A’;
```

The fail grade is given if the marks fall below 50, and execution continues on to the next statement after the ‘if’ construct (the whole code above is considered as one single compound statement). Otherwise, the mark is compared to 70, and ‘B’ grade is awarded if it is below 70, or ‘A’ grade if it is above or equal to 70. A fuller version with more grades will be shown later.

Example:

Show the output of the following code fragments:

```
int a = 1, b = 2;
if (a == 1)
if (b == 2)
    printf("***\n");
else
    printf("###\n");
```

```
int a = 1, b = 2;
if (a == 1)
    if (b == 2)
       printf("***\n");
    else
       printf("###\n");
```

```
  if (a == 1) {
     if (b == 2)
        printf("***\n");
  }
  else
     printf("###\n");
```

Let’s look at another example. The code below is wrong (division-by-zero error):

```
 if (a == 0)
     if (b == 0)
          printf("Both a and b are zeros.\n");
     else
          c = b/a;
```

The ‘else’ is attached to the second (nearer) ‘if’, and so when a is zero and b non-zero, the statement c = b/a will be carried out, resulting in a division-by-zero error.

The correct code should be:

```
  if (a == 0) {
     if (b == 0)
        printf("Both a and b are zeros.\n");
  }
  else
     c = b/a;
```

This code will not result in a division-by-zero error since the statement c = b/a will only be executed when a is non-zero.

Conditional Operator (?:)

C provides the conditional operator which is the only ternary operator that takes three operands. The operator with the operands forms a conditional expression with this syntax:

condition ? *expr1* : *expr2*

The first operand is the condition, the second operand is the value if the condition is true, and the third operand is the value if the condition is false.

For example,

`max = (a > b ? a : b);`   This is equivalent to:

```
if (a > b)
   max = a;
else
   max = b;
```

Similarly, the two pieces of code below are equivalent:

```
  if (marks < 50)
     printf("Failed\n");
  else
     printf("Passed\n");
```

AND …

```
  printf("%s\n", grade < 50 ? "Failed" : "Passed");
```

The ‘switch’ Construct

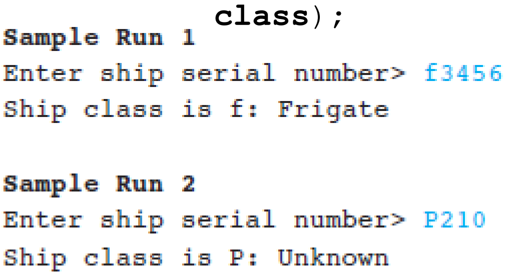
The ‘switch’ is a multi-way selection statement generalising the ‘if-else’ statement. The syntax is as follows:

```
     switch (expression) {
     case  v1: s1 ;
               break;
     case  v2: s2 ;
               break;
          . . .
     default:  sn ;
               break;   /* optional break */
     }
```

The value of the *expression* should be of type **int** or **char** ONLY.
After the expression is evaluated, control jumps to the appropriate ‘case’ label, and the statements in s_i are executed. Usually, the last statement before the next case is a ‘break’ statement. If there is no ‘break’ statement, the execution will fall through to the next statement in the succeeding case. There may be at most one default label in a ‘switch’ construct. The purpose of the default case is to capture cases that are not enumerated.

Here is an example:

```
#include <stdio.h>
int main(void)
{
     char class; /* input-character indicating class of ship */
     /* Read first character of serial number */
     printf("Enter ship serial number> ");
     scanf("%c", &class); /* scan first letter */
     /* Display first character followed by ship class */
     printf("Ship class is %c: ", class);
      switch (class) {
          case 'B':
          case 'b': printf ("Battleship\n");
                    break;
          case 'C':
          case 'c': printf ("Cruiser\n");
                    break;
          case 'D':
          case 'd': printf ("Destroyer\n");
                    break;
          case 'F':
          case 'f': printf ("Frigate\n");
                    break;
          default : printf ("Unknown ship class %c\n",
                            class);
       }
return (0);
}
```



```
Sample Run 1
Enter ship serial number> f3456
Ship class is f: Frigate

Sample Run 2
Enter ship serial number> P210
Ship class is P: Unknown
```

Chapter 4: Repetition Structure (Loops)

A loop is a group of instructions that is repeatedly executed while some condition stays true. An infinite loop is one that goes on forever, and that must be avoided.

There are basically two (general) types of loop control:

- *counter-controlled repetition* where the number of times of repetition is known beforehand,
- *sentinel-controlled repetition* where the loop stops when a special sentinel value is encountered, or when the terminating condition is met.

C provides 3 loop constructs:

- while loop
- for loop
- do-while loop

The 'while' Construct

The 'while' statement is a *pre-test* condition-controlled loop construct. It has the form:

```
while (expression)
     statement ;
```

The *expression* is the loop condition and it is evaluated. If it is non-zero (true), the *statement* in the loop body is executed and control is then passed back to the beginning of the loop, and the condition tested again. If it is zero (false), then the loop terminates and control is passed to the next statement after the 'while' construct. It is possible that the loop body is not executed at all, if the loop condition is false at entry.

As usual, the loop body could be a compound statement, in which case it must be enclosed in braces.

The code below prints n asterisks across the screen.

```
count_star = 0;
while (count_star < n) {
   printf("*");
   count_star++;
}
```

Assuming that *n* is an integer variable that contains the value 5, the above code prints 5 *'s. We say that the loop goes through 5 iterations. Trace the code.

The code could be rewritten as follows:

```
count_star = 0;
while (count_star++ < n)
   printf("*");
```

Example: Write a C program that computes the sum of the first 100 positive integers.

```
int num = 1;     /* declaration and */
int total = 0;   /* initialisation  */
while (num <= 100) {
    total += num;
    num++;
}
```

If the num++ statement is removed, and the condition (num <=100) is changed to (num++ <=100), following the previous example, would the amended code compute the same result?

Which of the following is/are equivalent to the example above?

`int num = 1;`
`int total = 0;`
`while (num <= 100)`
`{`
`   total += num;`
`   ++num;`
`}`
	`int num = 1;`
	`int total = 0;`
	`while (num <=100)`
	`   total+= num++;`
		`int num = 1;`
		`int total = 0;`
		`while (num <=100)`
		`   total+= ++num;`

The variable num above serves as the *loop control variable* in the loop statement. The loop control variable determines whether the loop should continue or terminate. The following operations involving the loop control variable are required for a successful loop construct:

- **Initialisation (*Start from*)** – before the loop is entered, the loop control variable must be initialised.
- **Testing (*Stop at*)** – condition involving the loop control variable is tested before the start of each loop iteration; if condition is true, loop body is executed.
- **Updating (*Step*)** – loop control variable is updated during each iteration (usually at the beginning or the end of the loop body).

Counter-control repetition and Sentinel-control repetition

When the number of iterations is known, a counter-control method is more appropriate. In this method, a counter is used to keep track of the number of iterations.

Sentinel-control is a more general approach. It is handy when the number of iterations cannot be determined beforehand, such as reading a list of values without knowing how many values are there. *A special value*, one that does not fall within the range of the data values, might be chosen to indicate the end of the list.

The code below illustrates counter-control repetition in summing a list of 10 values entered by the user:

```
#define N 10
. . .
total = 0;
count = 1;
while (count++ <= N) {
   printf("Enter score: ");
   scanf("%d", &score);
   total += score;
}
avg = (float) total / N;
printf("Average is %.2f\n", avg);
```

The code below illustrates sentinel-control repetition. The sentinel value of –1 is chosen to indicate the end of the list of scores.

```
#define SENTINEL -1
. . .
total = 0;
count = 0;

printf("Enter a score, OR -1 to end: ");
scanf("%d", &score);

while (score != SENTINEL) {
   total += score;
   count++;
   printf("Enter a score, OR -1 to end: ");
   scanf("%d", &score);
}
if (count) {
   avg = (float) total/count;
   printf("Average is %.2f\n", avg);
}
else
   printf("No scores were entered\n");
```

The disadvantage of user-defined sentinel such as –1 above is that the chosen sentinel value must not be a valid data. Hence, the range of valid values must be known first. If this is not possible, then setting an arbitrary sentinel is taking a risk. The more general solution is to use the *system-defined end-of-file (EOF)* character (Will be discussed later in this chapter).

The ‘do-while’ Construct

The ‘do-while’ statement is a loop structure with a *post-test* condition, in contrast to the pre-test condition in the ‘while’ statement. A post-test condition loop structure means that the condition is tested after the loop body. This implies that *the loop body is executed at least once*.

The syntax for the ‘do-while’ construct is as follows:

```
do
     statement ;
while (expression);
```

Below are two examples that illustrate the use of ‘do-while’ constructs.

```
c = 1;
do {
    printf ("%d ", c);
}while (++count <= 10);
```

```
do {
   printf("Enter a letter A through E: ");
   scanf("%c", &letter);
  } while (letter < 'A' || letter > 'E');
```

The second example above shows the common technique to request for data repeatedly until it is valid.

Flag-controlled Loops

When the loop condition becomes too complex, flags may be used to simplify the code, and makes it easier to read and understand. Here is an example of a flag-controlled loop:

```
valid = 1;
while (valid) {
   printf("Enter a letter A thru E: ");
   scanf("%c", &letter);
   valid = (letter >= 'A' && letter <= 'E');
}
```

The 'for' Construct

The 'for' statement is another pre-test condition-control loop structure. It provides a more compact form for counter-controlled loops.

The syntax of the 'for' construct is as follows:

```
for(initialization; stopping-condition; update-expression )

   statement ;
```

The 'for' construct is equivalent to this 'while' construct:

```
   initialisation;

   while (stopping-condition) {
        statement;
        update-expression;
   }
```

An example:

```
for (count_star = 0;  /* init      */
       count_star < N;  /* condition */
       count_star++)    /* update    */
     printf ("*");
```

The *initialisation-expression* and *update-expression* are often comma-separated lists of expressions. The comma operator evaluates the list from left to right. For example,

```
for (x = 1, total = 0; x <= 100; x++)
   total += x;
```

In this case, the initialisation-expression consists of two expression, namely, x = 1 and total = 0.

Any of the three expressions in the 'for' header can be omitted, but the semi-colons must remain. If the *initialisation-expression* is missing, then the loop does not perform any initialisation steps before executing the loop body. You have to ensure that all essential initialisation steps are done before the loop, as shown here:

```
x = 1;
total = 0;
for (; x <= 100; x++)
   total += x;
```

If the *update-expression* is missing, the loop does not perform any update operation. Again, you must ensure that the necessary update operations are done in the loop body.

```
for (x = 1, total = 0; x <= 100;) {
   total += x;
   x++;
}
```

If the *loop-condition* is missing, then the test is always true. This means that the following loop is infinite:

```
for (x = 1, total = 0; ; x++)

   total += x;
```

Common Mistakes

Here are a few common mistakes when it comes to programming loops.

Adding a semi-colon at the wrong place. For instance, in this code:

```
for (x = 1; x <= 10; x++);
   printf("%d\n", x);
```

as well as in this code:

```
x = 1;
while (x <= 10);
   printf("%d\n", x++);
```

the semi-colons at the end of the ‘for’ line and the ‘while’ line create empty statements, and so the printf() statements no longer fall inside the loop body. *What output do the above two codes produce?*

Omitting semi-colons in the ‘for’ header, and mixing up semi-colons with commas, are also quite common.

Another common mistake is to execute the loop one more time or one fewer time than desired. For example, the following code executes n+1 times (must be from 0 to n-1):

```
for (count = 0; count <= n; ++count)
   sum += count;
```

The ‘break’ and ‘continue’ Statements

The ‘break’ and continue statements are used to alter the flow of control.

The ‘break’ statement: terminates a loop under some special condition

The ‘continue’ statement: skips a section of the loop body in an iteration.

The ‘break’ statement in a ‘switch’, ‘while’, ‘do-while’ or ‘for’ structure causes immediate exit from the structure. The example below demonstrates the effect of a ‘break’ statement in a loop body.

```
#include <stdio.h>
main()
{
   int x;
   for (x = 1; x <= 10; x++) {
      if (x == 5)
         break;    /* break loop only if x == 5 */
      printf("%d ", x);
   }
   printf("\nBroke out of loop at x == %d\n", x);
   return 0;
}
```

The code produces the output below:

```
1 2 3 4
Broke out of loop at x == 5
```

The ‘continue’ statement in a ‘while’, ‘do-while’ or ‘for’ structure skips the remaining statements in the body, to the next iteration of the loop, as shown below:

```
#include <stdio.h>
main()
{
   int x;
   for (x = 1; x <= 10; x++) {
      if (x == 5)
         continue;  /* skip remaining code in
                       loop only if x == 5 */
      printf("%d ", x);
   }
   printf("\nUsed 'continue' to ");
   printf("skip printing the value 5\n");
   return 0;
}
```

The output for this code is:

```
1 2 3 4 6 7 8 9 10
Used 'continue' to skip printing the value 5
```

Example :- Write a C program to simulate a basic calculator.

```
#include <stdio.h>
int main()
{
     double num1, num2, result ;
     char op ;
     int flag, cont = 1;
     while ( cont )
     {
          printf ( " Enter number operator number\n" ) ;
         scanf ("%lf %c %lf", &num1, &op, &num2 ) ;
         flag = 1;
         switch ( op )
         {
                case '+' :  result = num1 + num2 ;
```

```
                                    break ;
                case '-' :   result = num1 - num2 ;
                                    break ;
                case '*' :  result = num1 * num2 ;
                                    break ;
                case '/' :  if ( num2 != 0.0 ) {
                                 result = num1 / num2 ;
                        break ;
        }
          // else we allow to fall through for error message
                default : {
                           printf ("ERROR -- Invalid operation or
                                     division by 0.0\n" ) ;
                           flag = 0;
                    }
          }
          if (flag)
            printf( "%f %c %f = %f\n", num1, op, num2, result) ;

        printf("Do you want to continue? enter 1 or 0 to exit\n");
        scanf("%d",&cont);
       }   /* while statement  */
     return 0;
}
```

NB : The break statement need not be included at the end of the case statement body if it is logically correct for execution to fall through to the next case statement (as in the case of division by 0.0) or to the end of the switch statement (as in the case of default :).

Nested Loops and Combined Structures

Just as we have nested ‘if’ and ‘if-else’ statements, we could also build nested loops, or combine selection structures with repetition structures.

The following code use a nested ‘for’ loop to print a pattern of asterisks:

```
  for (i = 1; i <= 4; ++i) {
     for (j = 1; j <= 6; ++j)
        printf("*");
     printf("\n");
  }
```

The output is 4 rows by 6 columns of *’s:

```
******
******
******
******
```

What about this code?

```
  for (i = 1; i <= 4; ++i) {
     for (j = 1; j <= i; ++j)
        printf("*");
     printf("\n");
  }
```

The output this time is:

```
*
**
***
****
```

Another example:

```
 for (i = 1; i <= 6; ++i) {
    if (i <= 3)
      printf("%d", i);
    else
       for (j = 1; j <= i; ++j)
          printf("*");
    printf("\n");
 }
```

The output is:

```
1
2
3
****
*****
******
```

Knowing how to combine control structures is an essential skill in programming and solving algorithmic problems. You need to have plenty of practice on this.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a=50;
    int i;
    for (i=2; i<=a;i+=2)
    {
        printf("%5d",i);

        if (i%5==0)

            printf ("\n");
    }
    return 0;
}
```

```
    2    4    6    8   10
   12   14   16   18   20
   22   24   26   28   30
   32   34   36   38   40
   42   44   46   48   50
Program ended with exit code: 0
```

```
#include <stdio.h>
#include <stdlib.h>
int isPrime(int x);
int main()
{
    for (int i=2;i<=50;i++)
        if(isPrime(i))
            printf("%d\t",i);
    return 0;
}
int isPrime(int x){
    int prime = 1;
    for(int j=2;j<x;j++)
        if(x%j==0)
            prime=0;
    return prime;
}
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
Program ended with exit code: 0
```

```
#include<stdio.h>

int main()
{
    int n, c, k;
    printf("Enter number of rows\n");
    scanf("%d",&n);
    for ( c = 1 ; c <= n ; c++){
        for ( k = 1 ; k <= c ; k++ )
            printf("*");

        printf("\n");
    }
    for ( c = n - 2 ; c >= 0 ; c-- ){
        for ( k = c ; k >= 0 ; k-- )
            printf("*");
        printf("\n");
    }
    return 0;
}
```

```
Enter number of rows
9
*
**
***
****
*****
******
*******
********
*********
********
*******
******
*****
****
***
**
*
Program ended with exit
code: 0
```

```c
#include<stdio.h>

int main(){
    int row, c, n, temp;
    printf("Enter the number of rows in
pyramid of stars you wish to see ");
    scanf("%d",&n);
    temp = n;
    for ( row = 1 ; row <= n ; row++ ){
        for ( c = 1 ; c < temp ; c++ )
            printf(" ");
        temp--;
        for ( c = 1 ; c <= 2*row - 1 ; c++ )
            printf("*");
        printf("\n");
    }
    return 0;
}
```

```
Enter the number of
rows in pyramid of stars
you wish to see 9
        *
       ***
      *****
     *******
    *********
   ***********
  *************
 ***************
*****************
Program ended with exit
code: 0
```

```c
#include<stdio.h>

int main()
{
    int n, c, k, space, count = 1;
    printf("Enter number of rows\n");
    scanf("%d",&n);
    space = n;
    for ( c = 1 ; c <= n ; c++){
        for( k = 1 ; k < space ; k++)
            printf(" ");
        for ( k = 1 ; k <= c ; k++){
            printf("*");
            if ( c > 1 && count < c){
                printf("A");
                count++;
            }
        }
        printf("\n");
        space--;
        count = 1;
    }
    return 0;
}
```

```
Enter number of rows
5
    *
   *A*
  *A*A*
 *A*A*A*
*A*A*A*A*
Program ended with exit
code: 0
```

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
    char i;
    for (i = 'A'; i<='Z';i++){
        printf("%4c",i);
        if (i%4==0)
            printf("\n");
    }
    return 0;
}
```

```
  A  B  C  D
  E  F  G  H
  I  J  K  L
  M  N  O  P
  Q  R  S  T
  U  V  W  X
  Y  Z
Program ended with exit
code: 0
```

```c
#include<stdio.h>

int main(){
    int n, c, k;
    printf("Enter number of rows\n");
    scanf("%d",&n);
    for ( c = 1 ; c <= n ; c++ ){
        for( k = 1 ; k <= c ; k++ )
            printf("*");
        printf("\n");
    }
    return 0;
}
```

```
Enter number of rows
8
*
**
***
****
*****
******
*******
********
Program ended with exit
code: 0
```

```c
#include <stdio.h>
int main()
{
    int n, c, k = 2, j;
    printf("Enter number of rows\n");
    scanf("%d",&n);
    for ( j = 1 ; j <= n ; j++ ){
        for ( c = 1 ; c <= 2*n-k ; c++)
            printf(" ");
        k = k + 2;
        for ( c = 1 ; c <= j ; c++)
            printf("*   ");
        printf("\n");
    }
    return 0;
}
```

```
Enter number of rows
9
        *
       * *
      * * *
     * * * *
    * * * * *
   * * * * * *
  * * * * * * *
 * * * * * * * *
* * * * * * * * *
Program ended with exit
code: 0
```

End of file sentinel loop:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *inp;
    char c;
    int sum =0, score, input_status;
    inp = fopen("score.txt","r");
    printf ("Scores\n");
    input_status = fscanf(inp,"%d",&score);
    while (input_status ==1)
    {
        // printf("end of file is %2d\n",EOF);
        //printf("input_status is %2d\n",input_status);
        printf("%5d\n",score);
        sum+=score;
        input_status = fscanf(inp,"%d",&score);
    }
    if (input_status==EOF)
        printf ("\n Sum of the scores is %d\n\n",sum);
    else
    {
        fscanf(inp,"%c",&c);
        printf ("Error at %c",c);
    }
    //printf("EOF at the last line is %5d\n",EOF);
    //printf("input_status at the last line
is%5d\n",input_status);
    fclose (inp);
    return 0;
}
```

Exercises

1. Write a program which prints out the ASCII values of all characters input at the keyboard terminating only when the character `q' or `Q' is entered.

2. Write a program to keep count of the occurrence of a user specified character in a stream of characters of known length (e.g. 50 characters) input from the keyboard. Compare this to the total number of characters input when ignoring all but alphabetic characters.

4. Note: The ASCII values of 'A'...'Z' are 65...90 and 'a'...'z' are 97...122.

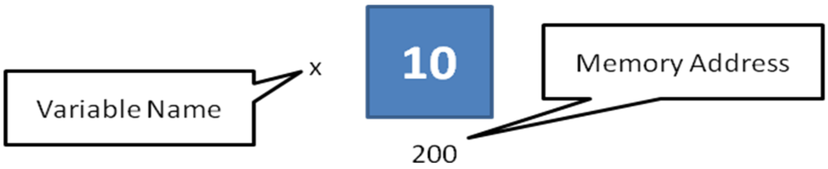
5. Write a program that allows the user to read a user specified number of double precision floating point numbers from the keyboard. Your program should calculate the sum and the average of the numbers input. *Try and ensure that any erroneous input is refused by your program, e.g. inadvertently entering a non-numeric character etc.*

6. Write a program which simulates the action of a simple calculator. The program should take as input two integer numbers then a character which is one of +,-,*,/,%. The numbers should be then processed according to the operator input and the result printed out.

Chapter 5: Pointers

A ***pointer*** is a ***variable*** that is used to store a ***memory address***. Most commonly the address is the location of another variable in memory.

If one variable holds the address of another then it is said to point to the second variable.
In the following example, x is an integer variable that stored in a memory location of address 200.



Another example:

Address	Value	Variable
1000		
1004	1012	ivar_ptr
1008		
1012	23	ivar
1016		

In the above illustration *ivar* is a variable of type int with a value 23 and stored at memory location 1012. *ivar_ptr* is a variable of type *pointer to int* which has a value of 1012 and is stored at memory location 1004. Thus ivar_ptr is said to point to the variable *ivar* and allows us to refer indirectly to it in memory.

NB : It should be remembered that *ivar_ptr* is a variable itself with a specific piece of memory associated with it, in this 32-bit case four bytes at address 1004 which is used to store an address.

Pointer Variables

Pointers like all other variables in C must be declared as such prior to use.

Syntax : **`type    *ptr_name ;`**

which indicates that *ptr* is a pointer to a variable of type *type*. For example

`int   *ptr ;`	declares a pointer `ptr` to variables of type `int`.
`char   *ch ;`	declares a pointer `ch` to variables of type `char`.
double *dblPtr	declares a pointer dblPtr to variables of type `double`.

Pointer Operators * and &

& is a unary operator that returns the **address** of its operand which must be a variable.

For Example :-

```
    int *m ;
    int count=125, i ;/* m is a pointer to int, count, i are
                        integers */
    m = &count ;
```

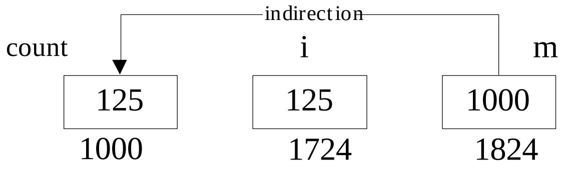
The address of the variable *count* is placed in the pointer variable *m*.

The * operator is the complement of the address operator **&** and is normally termed the indirection operator. Like the & operator it is a unary operator and it returns the **value** of the variable located at the address its operand stores.

For Example :-

```
    i = *m ;
```

assigns the value which is located at the memory location whose address is stored in m, to the integer i. So essentially in this case we have assigned the value of the variable count to the variable i. The final situation is illustrated below.



A pointer may also be initialised to 0 (zero) or NULL which means it is pointing at nothing.

NB : NULL is #defined in <stdio.h>.

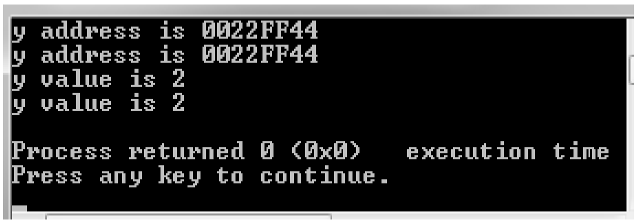
For Example :-

```
    int var1, var2 ;
    int *ptr1, *ptr2 = &var2 ;
    int *ptr3 = NULL ;
    ...
    ptr1 = &var1 ;
```

`ptr1` and `ptr2` are now pointing to data locations within the program so we are free to manipulate them at will i.e. we are free to manipulate the piece of memory they point to.

Pointers Examples:

```
#include <stdio.h>
int main() {
  int y=2;
  int *p;
  p = &y;
  printf ("y address is %p \n",&y);
  printf ("y address is %p \n‟,p);
  printf ("y value is %d \n",y);
  printf ("y value is %d \n",*p);
  return 0;
}
```



```
y address is 0022FF44
y address is 0022FF44
y value is 2
y value is 2

Process returned 0 (0x0)   execution time
Press any key to continue.
```

```
NB: *p = *(&y)=*(0022FF44)= y = 2
```

Like other variables, always initialize pointers before using them!!!
For example:

```
#include<stdio.h>
int main()
{
    int x;
    int *p;
    scanf("%d",p); /*Incorrect … the pointer is not
initialized*/
    p = &x;
    scanf("%d",p); /* Correct */

     printf("%d",x);
}
```

You can use pointers to access the values of other variables. To do this, use the * operator (dereferencing operator).

```
#include<stdio.h>
int main()
{
    int n, m=3, *p;
    p= &m;
    n=*p;
    printf ("%d\n", n); // prints 3
    printf("%d\n",*p); // also prints 3
}
```

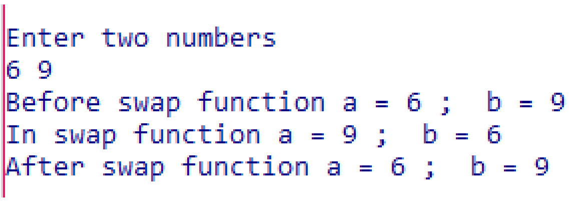
Another Example

```
#include<stdio.h>
int main()
{
    int m= 3, n=100, *p;
    p= &m;
    printf("m is %d\n",*p);
    m++;
    printf("now m is %d\n ",*p);
    p= &n;
    printf("n is %d\n",*p);
    *p= 500; /* *p is at the left of "= " */
    printf ("now n is %d\n ", n);
    return 0;
}
```

Pointers as Function Parameters (Call by Reference)

In the following example we want to exchange the values of two integer variables:

```
#include <stdio.h>
void swap( int, int);
int main( )
{
     int a, b ;
     printf( "Enter two numbers\n" ) ;
     scanf( "%d%d", &a, &b ) ;
     printf( "Before swap function a = %d ;  b = %d \n", a, b ) ;
     swap( a, b ) ;
     printf( "After swap function a = %d ;  b = %d \n", a, b ) ;
     return 0;
}
void swap ( int  a, int  b )
{
     int temp ;
     temp = a ;
     a = b;
     b = temp ;
    printf( "In swap function a = %d ;  b = %d \n", a, b ) ;

}
```



```
Enter two numbers
6 9
Before swap function a = 6 ;  b = 9
In swap function a = 9 ;  b = 6
After swap function a = 6 ;  b = 9
```

As you can see in the previous example values of *a* and *b* are not changed in the main, while they changed in the function!! SO: When we wanted to swap two values using a function we were unable to actually swap the calling parameters as the call by value standard was employed. The solution to the problem is to use call by reference which is implemented in C by using pointers as is illustrated in the following example.

```
#include <stdio.h>
void swap( int*, int*);
void main( )
{
     int a, b ;
     printf( "Enter two numbers" ) ;
     scanf( " %d %d ", &a, &b ) ;
     printf( "a = %d ;  b = %d \n", a, b ) ;
     swap( &a, &b ) ;
     printf( "a = %d ;  b = %d \n", a, b ) ;
}
void swap ( int  *ptr1, int  *ptr2 )
{
     int temp ;
     temp = *ptr2 ;
     *ptr2 = *ptr1 ;
     *ptr1 = temp ;
}
```

The `swap()` function is now written to take integer pointers as parameters and so is called in `main()` as

```
     swap( &a, &b ) ;
```

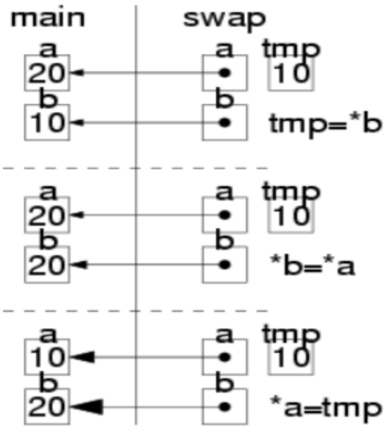
where the addresses of the variables are passed and copied into the pointer variables in the parameter list of swap(). These pointers must be de-referenced to manipulate the values, and it is values in the same memory locations as in main() we are swapping unlike the previous version of swap where we were only swapping local data values.

In our earlier call-by-value version of the program we called the function from main() as *swap(a,b);* and the values of these two calling arguments were copied into the formal arguments of function swap.

In our call-by-reference version above our formal arguments are pointers to int and it is the addresses contained in these pointers, (i.e. the pointer values), that are copied here into the formal arguments of the function. However when we de-reference these pointers we are accessing the values in the main() function as their addresses do not change.

You can see what is happening when you make a call to the swap function with a = 20 and b = 10.

```
swap( &a, &b ) ;
```

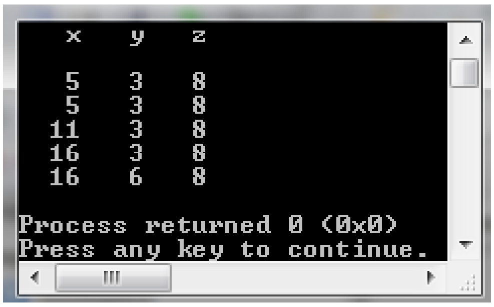


Another Eample:

```
#include <stdio.h>
void min_max(int, int, int*, int*);
int main()
{
     int x,y;
     int small,big;
     printf("Two integers: ");
     scan f (" %d %d" , &x , &y);
     min_max(x,y,&small,&big);
     printf("%d < = %d", small, big);
     return 0;
}

void min_max(int a, int b, int *min, int *max)
{
     if (a> b){
          *max= a;
          *min= b;
     }
     else{
          *max=b;
          *min=a;
     }
}
```

Example: give the output of the following program.

```
#include <stdio.h>
void sum(int a, int b, int *cp);
int main(void)
{
    int x, y, z;
    x = 5; y = 3;
    printf("   x   y   z \n\n");
    sum(x, y, &z);
    printf("%4d%4d%4d\n", x, y, z);
    sum(y, x, &z);
    printf("%4d%4d%4d\n", x, y, z);
    sum(z,y, &x);
    printf("%4d%4d%4d\n", x, y, z);
    sum(z, z, &x);
    printf("%4d%4d%4d\n", x, y, z);
    sum(y, y, &y);
    printf("%4d%4d%4d\n", x, y, z);
    return (0);
}
void sum(int a, int b, int *cp)
{
    *cp = a + b;
}
```



```
   x   y   z

   5   3   8
   5   3   8
  11   3   8
  16   3   8
  16   6   8

Process returned 0 (0x0)
Press any key to continue.
```

Chapter 6: Arrays

An array is a collection of variables of the same type that are referenced by a common name. Specific *elements* or variables in the array are accessed by means of an *index* into the array.

In C all arrays consist of contiguous memory locations. The lowest address (0) corresponds to the first element in the array while the largest address corresponds to the last element in the array.

C supports both single and multi-dimensional arrays.

Single Dimension Arrays

Syntax : **type array_name[size] ;**

where type is the type of each element in the array, var_name is any valid C identifier, and size is the number of elements in the array which has to be a constant value.

NB : In C all arrays use zero as the index to the first element in the array.

For Example :-

```
      int array[ 5 ] ;
```

which we might illustrate as follows for a 32-bit system where each int requires 4 bytes.

array[0]	12	loc^n 1000
array[1]	-345	loc^n 1004
array[2]	342	loc^n 1008
array[3]	-30000	loc^n 1012
array[4]	23455	loc^n 1016

NB: The valid indices for array above are 0 .. 4, i.e. 0 .. size – 1.
NB: The array name is a pointer to the first element in the array. Array => &Array[0]

Initialising Arrays

Arrays can be initialised at time of declaration in the following manner.

```
 type array[ size ] = { value list };
```

For Example :-

```
     int i[5] = {1, 2, 3, 4, 5 } ;
```

Or you can initialize each element individually.

```
     i[0] = 1, i[1] = 2, etc.
```

The size specification in the declaration may be omitted which causes the compiler to count the number of elements in the value list and allocate appropriate storage.

For Example :- `int i[ ]  =  { 1, 2, 3, 4, 5 } ;`

For Example :- To read an array with values 0 .. 99

```
     int x[100] ;
     int i ;
     for ( i = 0; i < 100; i++ )
          x[i] = i ;
```

Arrays should be viewed as just collections of variables so we can treat the individual elements in the same way as any other variables. For example, we can obtain the address of each one as follows to read values into the array

```
     for ( i = 0; i < 100; i++ ) {
          printf( "Enter element %d", i + 1 ) ;
          scanf( "%d\n", &x[i] ) ;
     }
```

NB : Note the use of the *printf* statement here. As arrays are normally viewed as starting with index 1 the user will feel happier using this so it is good policy to use it in “public”.

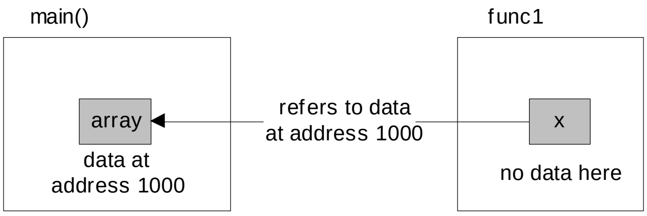
Arrays as arguments to functions

In C it is impossible to pass an entire array as an argument to a function -- instead the address of the array is passed as a parameter to the function. (In time we will regard this as a pointer).

For Example :-

```
 void main()
 {
   int array[20] ;

   func1( array ) ;/* passes pointer to array to func1 */
 }
```

Since we are passing the address of the array the function will be able to manipulate the actual data of the array in main(). This is call by reference as we are not making a copy of the data but are instead passing its address to the function. Thus the called function is manipulating the same data space as the calling function.



In the function receiving the array the formal parameters can be declared in one of three almost equivalent ways as follows :-

- As a sized array :

```
func1 ( int x[10] ) {
...
}
```

- As an unsized array :

```
func1 ( int x[ ] ) {
...
}
```

- As an actual pointer

```
func1 ( int *x ) {
...
}
```

All three methods are identical because each tells us that in this case the address of an array of integers is to be expected (passed from main).

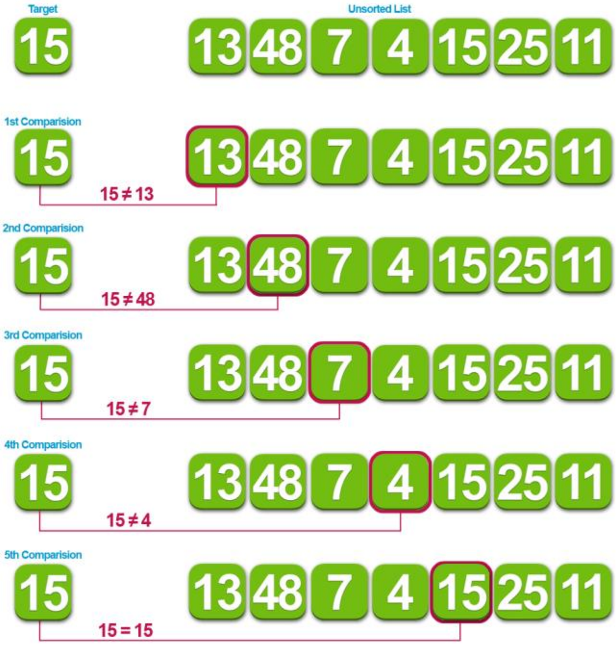
For Example :- Program to calculate the average value of an array of doubles.

```
#include <stdio.h>
void read_array( double array[ ], int size ) ;
double mean( double array[ ], int size ) ;
void main()
{
      double data[ 100 ] ;
      double average ;
      read_array( data, 100 ) ;
      average = mean( data, 100 ) ;
      printf("Average is %f \n",average);
}
void read_array( double array[ ], int size )
{
      int i ;
      for ( i = 0; i<100; i++ )      {
      printf( "\nEnter data value %d : i + 1 );
      scanf( "%lf", &array[i] ;
}
double mean( double array[ ], int size )
{
      double total = 0.0 ;
      int count = size ;
```

```
     while ( count-- )  // size is a local variable which we
can
   // use at will
     total += array[ count ] ;
     return ( total / size ) ;
}
```

Searching an element in the array (Linear Search):

Given a one-dimensional array and some value x, check wither x is in the array or not. If the element is found, the position (i.e., "index") of x in the array is returned, or -1 if x does not appear in the array. One option would be to use a **linear search**. In a linear search, we compare x (which we call the "key") with each element in the array list, starting at one end and progressing to the other. Graphically, we can imagine the following comparisons being made:



Function:

```
int Linear_search (int a[], int size, int key)
{
    int i;
   for(i=0;i<size;i++)
         if(a[i]==key)
             return i;
   return -1;
}
```

Sorting an Array

Selection Sort:

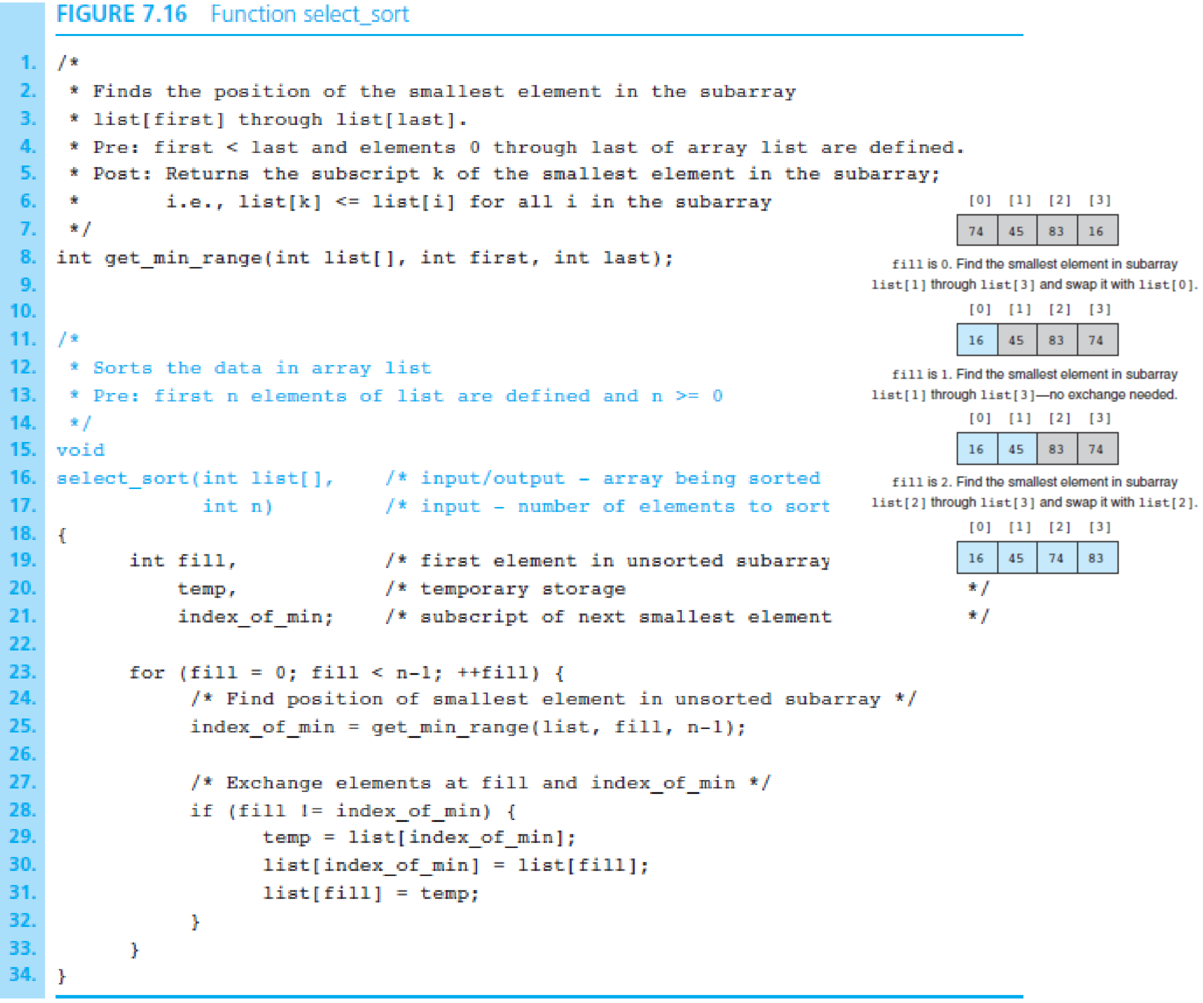


FIGURE 7.16 Function select_sort

```
/*
 * Finds the position of the smallest element in the subarray
 * list[first] through list[last].
 * Pre: first < last and elements 0 through last of array list are defined.
 * Post: Returns the subscript k of the smallest element in the subarray;
 *       i.e., list[k] <= list[i] for all i in the subarray
 */
int get_min_range(int list[], int first, int last);


/*
 * Sorts the data in array list
 * Pre: first n elements of list are defined and n >= 0
 */
void
select_sort(int list[],    /* input/output - array being sorted
            int n)         /* input - number of elements to sort
{
      int fill,            /* first element in unsorted subarray
          temp,            /* temporary storage                      */
          index_of_min;    /* subscript of next smallest element     */

      for (fill = 0; fill < n-1; ++fill) {
           /* Find position of smallest element in unsorted subarray */
           index_of_min = get_min_range(list, fill, n-1);

           /* Exchange elements at fill and index_of_min */
           if (fill != index_of_min) {
                 temp = list[index_of_min];
                 list[index_of_min] = list[fill];
                 list[fill] = temp;
           }
      }
}
```

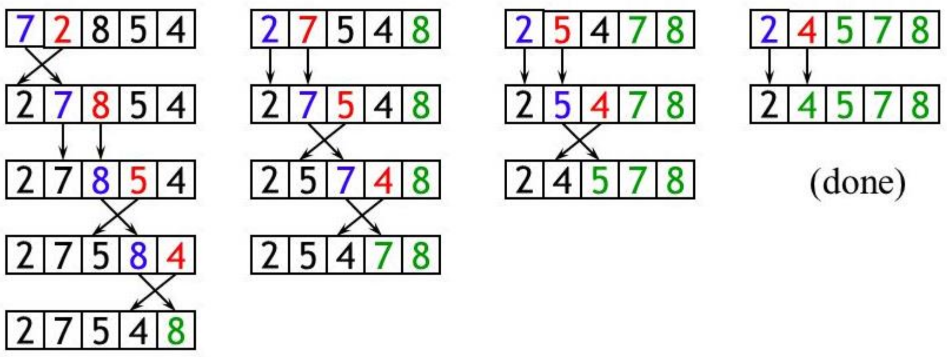
fill is 0. Find the smallest element in subarray list[1] through list[3] and swap it with list[0].

fill is 1. Find the smallest element in subarray list[1] through list[3]—no exchange needed.

fill is 2. Find the smallest element in subarray list[2] through list[3] and swap it with list[2].

Bubble Sort

Bubble Sort is a sorting algorithm which compares two adjacent elements and swap them if they are not in the right order. To sort the entire array, the array is traversed n-1 time (array having n elements). These are called passes, In the first pass the largest element moves to the last position (sorting in ascending order). So if the original (unsorted) array is:



```
void bubble_sort(int array[], int n){
   int temp, j, i;
   for (i = 0; i<n-1; i++){
      for ( j=i+1; j< n; j++)
         if (array[i]  < array[j] )
         {
            temp = array[i];
            array[i] = array[j];
            array[j] = temp;
         }
  }
}
```

Array Example

Write a complete C program that does the following functionalities. Write a function for each one and call it in the main.

1. Fill array elements from the keyboard
2. Print array elements
3. Find the summation of two arrays (print sum array in main).
4. Find the maximum value in the array (print it in main)
5. Search the array for a specific key.
6. Sort the array in ascending order.

```
#include <stdio.h>
#include <stdlib.h>
void read_array(int a[], int s);
void print_array (int a[], int s);
void sum_arr(int a1[],int a2[], int aSum[],int s);
int find_Max(int ar[], int s);
int search_arr(int ar[], int s, int k);//searching array
void sort_array(int a[], int s);
int main(){
    int size;
    printf ("Please enter the size of the array\n");
    scanf("%d",&size);
    int arr[size];
    int arr2[size];
    int sum_array[size];
    printf("\nEnter Array 1 elements\n\n");
    read_array(arr,size);
    printf("\nEnter Array 2 elements\n\n");
    read_array(arr2,size);
    sum_arr (arr,arr2,sum_array,size);
//*******************************************
    printf("Sum Array: \n");
    print_array (sum_array, size);
//*******************************************
    int max = find_Max(arr,size);
    printf("\nThe Max element in array 1 is %d\n",max);
//*******************************************
    int key;
    printf("\nPlease enter the key you want to search\n");
    scanf("%d",&key);
    int found = search_arr(arr, size, key);
    if (!found)
        printf("%d has not been found\n",key);
    else
        printf("%d has been found %d times\n",key, found);
//*******************************************
    sort_array(arr, size);
    printf("\nThe sorted array (1) (in descending order) is \n\n");
    print_array (arr, size);
   return 0;
}
void read_array(int a[], int s){
    int i;
    for (i = 0; i< s; i++){
        printf("Please enter element number %d: ",i+1);
        scanf("%d",&a[i]);
    }
}
void print_array (int a[], int s){
    int i;
    printf("[ ");
    for (i = 0; i< s; i++)
        printf("%d ",a[i]);
    printf("]\n");
}
void sum_arr(int a1[], int a2[], int aSum[],int s){
    int i;
    for (i = 0; i< s; i++)
        aSum[i] = a1[i]+a2[i];
}
int find_Max(int ar[], int s){
    int max = ar[0],i;
    for (i = 1; i< s; i++)
        if(ar[i]> max)
            max=ar[i];
    return max;
}
int search_arr(int ar[], int s, int k){
    int i, count =0;
    for (i = 0; i< s; i++)
        if(ar[i] == k)
           count++;
    return count;
}
void sort_array(int array[], int n){
    int temp, j, i;
    for (i = 0; i<n-1; i++){
        for ( j=i+1; j< n; j++)
            if (array[i]  < array[j] ){
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
   }
}
```

Multidimensional Arrays

Multidimensional arrays of any dimension are possible in C but in practice only two or three dimensional arrays are workable. The most common multidimensional array is a two dimensional array for example the computer displays, board games, a mathematical matrix etc.

Syntax : **`type   name [ rows ] [ columns ]  ;`**

For Example :- 2D array of dimension 2 X 3.

```
int d[ 2 ] [ 3 ] ;
```

d[0][0]	d[0][1]	d[0][2]
d[1][0]	d[1][1]	d[1][2]

A two dimensional array is actually an array of arrays, in the above case an array of two integer arrays (the rows) each with three elements, and is stored row-wise in memory.

For Example :- Program to fill in a 2D array with numbers 1 to 6 and to print it out row-wise.

```
#include <stdio.h>
void main( )
{
    int i, j, num[2][3] ;
    for ( i = 0; i < 2; i++ )
        for ( j = 0; j < 3; j ++ )
            num[i][j] =  i * 3 + j + 1 ;
    for ( i = 0; i < 2; i++ )
    {
        for ( j = 0; j < 3; j ++ )
            printf("%d ",num[i][j]  ) ;
        printf("\n" );
    }
}
```

For Example :- Program to tabulate ***sin(x)*** from x = 0 to 1 radians in steps of 0.1 radians.

```
#include <stdio.h>
#include <math.h>
int main()
{
    int i  ;
    double x ;
    double table[20][2] ;// we will need 100 data points for
                // the above range and step size and
    // will store both x and f(x)
    for ( x = 0.0, i = 0; x < 1.0; x += 0.1, i++ )
   {
      table[i][0] = x ;
```

```
         table[i][1] = sin( x ) ;
         printf("\n Sin( %lf ) = %lf", table[i][0], table[i][1] );
      }
      return 0;
}
```

To initialise a multidimensional array all but the leftmost index must be specified so that the compiler can index the array properly.

For Example :-

```
      int d[ ] [ 3 ] = { 1, 2, 3, 4, 5, 6 } ;
```

However it is more useful to enclose the individual row values in curly braces for clarity as follows.

```
      int d[ ] [ 3 ] = { {1, 2, 3}, {4, 5, 6} } ;
```

Passing Multidimensional Arrays to a Function

Function calls with multi-dimensional arrays will be the same as with single dimension arrays as we will still only pass the address of the first element of the array.

However, to declare the formal parameters to the function we need to specify all but one of the dimensions of the array so that it may be indexed properly in the function.

For Example :-

2D array of doubles :- `double x[10][20] ;`

Call func1 with x a parameter :- `func1( x ) ;`

Declaration in func1 :-

```
func1( double y[ ][20] ) {
    ...
}
```

The compiler must at least be informed how many columns the matrix has to index it correctly. For example to access element y[5][3] of the array in memory the compiler might do the following

```
 element No =  5 * 20 + 3 = 103.
```

NB : Multi-dimensional arrays are stored row-wise so `y[5][3]` is the 4th element in the 6th row.

Since we are dealing with an array of doubles this means it must access the memory location 103 X 8 bytes from the beginning of the array.

Thus the compiler needs to know how many elements are in each row of the 2D array above. In general the compiler needs to know all dimensions except the leftmost at the very least.

For Example :- Program to add two 2 x 2 matrices.

```
#include <stdio.h>
void mat_read( int mat[2][2] ) ; // Write these two functions for yourself
void mat_print( int mat[2][2] ) ;
void mat_add( int mat1[ ][2], int mat2[ ][2], int mat3[ ][2] ) ;
int main()
{
    int mat_a[2][2], mat_b[2][2], mat_res[2][2] ;
      puts( "Enter Matrix a row-wise :-\n" );
      mat_read( mat_a ) ;
      puts( "\nMatrix a is :-\n") ;
      mat_print( mat_a ) ;
      puts( "Enter Matrix b row-wise" );
      mat_read( mat_b ) ;
      puts( "\nMatrix b is :-\n" ) ;
      mat_print( mat_b ) ;

      mat_add( mat_a, mat_b, mat_res ) ;

      puts( "The resultant matrix is\n" ) ;
      mat_print( mat_res ) ;
      return 0;
}

void mat_add( int mat1[ ][2], int mat2[ ][2], int mat_res[ ][2] )
{
      int j, k ;
      for ( j = 0; j < 2; j++ )
            for ( k = 0; k < 2; k++ )
                  mat_res[j][k] = mat1[j][k] + mat2[j][k]  ;
}
```

Pointers and Arrays

There is a very close relationship between pointer and array notation in C. As we have seen already the name of an array is actually the address in memory of the array and so it is essentially a constant pointer.

For Example :-

```
char str[80], *ptr ;

ptr =  str ;/* causes ptr to point to start of string str  */
ptr = &str[0] ;     /* this performs the same as above */
```

It is illegal however to do the following

```
str = ptr ;    /* illegal */
```

as `str` is a constant pointer and so its value i.e. the address it holds cannot be changed.

Instead of using the normal method of accessing array elements using an index we can use pointers in much the same way to access them as follows.

```
char str[80], *ptr , ch;

ptr = str ;          // position the pointer appropriately

*ptr = 'a' ;         // access first element i.e. str[0]
ch = *( ptr + 1 )  ;     // access second element i.e. str[1]
```

Thus ***(array + index)** is equivalent to **array[index]**.

Note that the parentheses are necessary above as the precedence of * is higher than that of +. The expression

```
     ch = *ptr + 1 ;
```

for example says to access the character pointed to by ptr (str[0] in above example with value ‘a’) and to add the value 1 to it. This causes the ASCII value of ‘a’ to be incremented by 1 so that the value assigned to the variable ch is ‘b’.

```
     char ch[3]={'A','B','C'}, *p;
     p = ch;
     printf("%c\n",*p); // A
     printf("%c\n",*p+1); // B
    printf("%c\n",*(p+1)); // B
```

In fact so close is the relationship between the two forms that we can do the following

```
int x[10], *ptr ;

ptr = x ;
ptr[4] = 10 ;  /* accesses element 5 of array by indexing a pointer */
```

Pointer Arithmetic

Pointer variables can be manipulated in certain limited ways. Many of the manipulations are most useful when dealing with arrays which are stored in contiguous memory locations. Knowing the layout of memory enables us to traverse it using a pointer and not get completely lost.

- **Assignment**

```
int count, *p1, *p2 ;

p1 = &count ;     // assign the address of a variable directly
p2 = p1 ;         // assign the value of another pointer variable, an address
```

- **Addition / Subtraction**

The value a pointer holds is just the address of a variable in memory, which is normally a four byte entity. It is possible to modify this address by integer addition and subtraction if necessary. Consider the following we assume a 32-bit system and hence 32-bit integers.

```
int *ptr ;
int array[3] = { 100, 101,
102 } ;
ptr = array ;
```

	Address	Value
ptr	1000	2008
	⍰	⍰
array[0]	2008	100
array[1]	2012	101
array[2]	2016	102

We now have the pointer variable ptr pointing at the start of *array* which is stored at memory location 2008 in our illustration. Since we know that element array[1] is stored at address 2012 directly after element array[0] we could perform the following to access its value using the pointer.

```
ptr += 1 ;
```

This surprisingly will cause ptr to hold the value 1012 which is the address of array[1], so we can access the value of element array[1]. The reason for this is that ptr is defined to be a pointer to type int, which are four bytes in size on a 32-bit system. When we add 1 to ptr what we want to happen is to point to the **next integer** in memory. Since an integer requires four bytes of storage the compiler increments ptr by 4. Likewise a pointer to type char would be incremented by 1, a pointer to float by 4, etc.

Similarly we can carry out integer subtraction to move the pointer backwards in memory.

```
ptr = ptr - 1 ;
ptr -= 10 ;
```

The shorthand operators ++ and -- can also be used with pointers. In our continuing example with integers the statement ***ptr++ ;*** will cause the address in ptr to be incremented by 4 and so point to the next integer in memory and similarly ***ptr-- ;*** will cause the address in ptr to be decremented by 4 and point to the previous integer in memory.

NB : Two pointer variables may not be added together (it does not make any logical sense).

```
char *p1, *p2 ;
p1 = p1 + p2 ; /* illegal operation */
```

Two pointers may however be subtracted as follows.

```
int *p1, *p2, array[3], count ;
p1 = array ;
p2 = &array[2] ;

count = p2 - p1 ;    /* legal */
```

The result of such an operation is not however a pointer, it is the number of elements of the base type of the pointer that lie between the two pointers in memory.

Chapter 6: Strings

In C a string is defined as a character array which is terminated by a special character, the null character '\0', as there is no string type as such in C.

Thus the string or character array must always be defined to be one character longer than is needed in order to cater for the '\0'.

For Example :- string to hold 5 characters

```
 char s[6] ;
```

					'\0'

A string constant is simply a list of characters within double quotes e.g. "Hello" with the '\0' character being automatically appended at the end by the compiler.

A string may be initialised as simply as follows

```
 char s[6] = "Hello" ;
```

'H'	'e'	'l'	'l'	'o'	'\0'

as opposed to

```
        char s[6] = { 'H', 'e', 'l', 'l', 'o', '\0' } ;
```

Again the size specification may be omitted allowing the compiler to determine the size required.

Manipulating Strings

We can print out the contents of a string using printf() as we have seen already or by using puts().

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

Strings can be read in using scanf()

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

where we do not require the familiar & as **the name of an array without any index or square braces is also the address of the array**.

A string can also be read in using gets() // scanf will read the string till the first space!!!

```
        gets ( s ) ;
```

printf and scanf with strings

scanf and printf handle the strings as long as the %s available:

```
char str[6]="Hello";
printf("%s\n",str);
```

If the number of characters (n) is less than the string length (m) then the first *n* characters (from 0 to n-1) will be filled and the character (n) will be filled by the null character (\n).

```
char str[10]="Hello";
printf("%s\n",str);
```

'H'	'e'	'l'	'l'	'o'	'\0'				

<u>Printf</u>

```
char str[6]="Hello";
printf("%8s\n",str); // %8s would print the string right align
```

			H	e	l	l	o

```
char str[6]="Hello";
printf("%-8s\n", str); //  %-8s would print the string left
align
```

H	e	l	l	o			

```
char one_string[4];
one_string = "Hi";
```

|error: incompatible types in assignment|

String Manipulation Functions

- char *strcpy (char *dest, char *src) - Copy src string string into dest string.

```
#include <stdio.h>
#include <strings.h>
int main() {
  char input_str[20];
  char *output_str;

  strcpy(input_str, "Hello");
  printf("input_str: %s\n", input_str);

  output_str = strcpy(input_str, "World");

  printf("input_str: %s\n", input_str);
  printf("output_str: %s\n", output_str);

  return 0;
}
```

It will produce the following result:

```
input_str: Hello
input_str: World
output_str: World
```

- char *strncpy(char *string1, char *string2, int n) - Copy first n characters of string2 to stringl .

```
#include <stdio.h>
#include <strings.h>
int main() {
    char input_str[20] = "Majdi";
    char *output_str;
    printf("input_str: %s\n", input_str);
    strncpy(input_str, "Wamy", 2);
    printf("input_str: %s\n", input_str);

    output_str = strncpy(input_str, "World", 3);

    printf("input_str: %s\n", input_str);
    printf("output_str: %s\n", output_str);

    return 0;
}
```

It will produce the following result:

```
input_str: Majdi
input_str: Wajdi
input_str: Wordi
output_str: Wordi
Program ended with exit code: 0
```

- int strcmp(char *string1, char *string2) - Compare string1 and string2 to determine alphabetic order.

```
Return Value
• if Return value if < 0 then it indicates string1 is less than
     string2

• if Return value if > 0 then it indicates string2 is less than
     string1

• if Return value if = 0 then it indicates string1 is equal to
     string2
```

```
#include <stdio.h>
#include <strings.h>
int main() {
    char string1[20];
    char string2[20];

    strcpy(string1, "Majdi");
    strcpy(string2, "Majdi Mafarja");
    printf("strcmp( string1, string2) Return Value is : %d\n", strcmp(
string1, string2));

    strcpy(string1, "Majdi Mafarja");
    strcpy(string2, "Majdi");
    printf("strcmp( string1, string2) Return Value is : %d\n", strcmp(
string1, string2));

    strcpy(string1, "Majdi");
    strcpy(string2, "Majdi");
    printf("strcmp( string1, string2) Return Value is : %d\n", strcmp(
string1, string2));

    return 0;
}
```

It will produce the following result:

```
strcmp( string1, string2) Return Value is : -32
strcmp( string1, string2) Return Value is : 32
strcmp( string1, string2) Return Value is : 0
Program ended with exit code: 0
```

- int strncmp(char *string1, char *string2, int n) - Compare first n characters of two strings.

```
   #include <stdio.h>
   #include <strings.h>
   int main() {
       char string1[20];
       char string2[20];

       strcpy(string1, "Majdi");
       strcpy(string2, "Majdi Mafarja");
       printf("strcmp( string1, string2) Return Value is : %d\n",
strncmp( string1, string2,4));

    strcpy(string1, "Majdi Mafarja");
    strcpy(string2, "Majdi");
    printf("strcmp( string1, string2) Return Value is : %d\n",
strncmp( string1, string2,10));

    strcpy(string1, "Majdi");
    strcpy(string2, "Majdi");
    printf("strcmp( string1, string2) Return Value is : %d\n",
strncmp( string1, string2,20));

    return 0;
}
```

It will produce the following result:

```
strcmp( string1, string2) Return Value is : 0
strcmp( string1, string2) Return Value is : 32
strcmp( string1, string2) Return Value is : 0
Program ended with exit code: 0
```

- int strlen(char *string) - Determine the length of a string.

```
#include <stdio.h>
#include <strings.h>
int main() {
    char string1[20];
    char string2[20];

    strcpy(string1, "Majdi");
    strcpy(string2, "Majdi Mafarja");
    printf("String 1 length is %d\n",strlen(string1));
    printf("String 2 length is %d\n",strlen(string2));

    return 0;
}
```

It will produce the following result:

```
String 1 length is 5
String 2 length is 13
```

- char *strcat(char *dest, const char *src); - Concatenate string src to the string dest.

```
#include <stdio.h>
#include <strings.h>
int main() {
    char string1[20];
    char string2[20];

    strcpy(string1, "Majdi");
    strcpy(string2, "Mafarja");
    printf("Returned String : %s\n", strcat( string1,
string2 ));
    printf("Concatenated String : %s\n", string1 );

    return 0;
}
•
```

It will produce the following result:

```
Returned String : MajdiMafarja
Concatenated String : MajdiMafarja
Program ended with exit code: 0
```

- char *strncat(char *dest, const char *src, int n); - Concatenate n chracters from string src to the string dest.

```
#include <stdio.h>
#include <strings.h>
int main() {
    char string1[20];
    char string2[20];

    strcpy(string1, "Majdi");
    strcpy(string2, "Mafarja");

    printf("Returned String : %s\n", strncat( string1,
string2, 4 ));
    printf("Concatenated String : %s\n", string1 );

    return 0;
}
```

It will produce the following result:

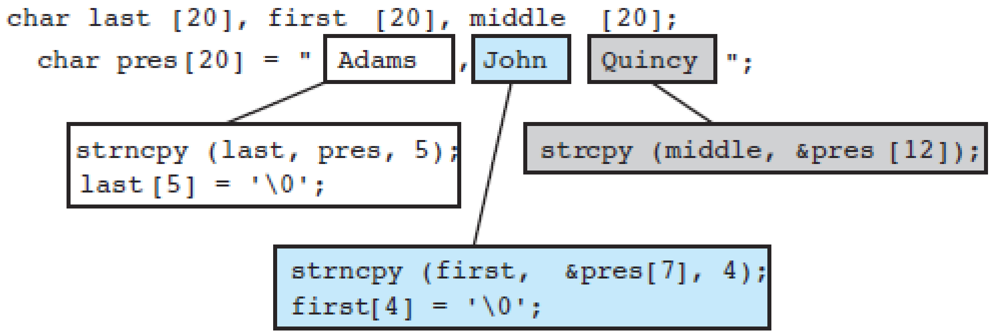
```
Returned String : MajdiMafa
Concatenated String : MajdiMafa
Program ended with exit code: 0
```

- char *strtok(char *s, const char *delim) - Parse the string s into tokens using delim as delimiter.

```
#include <stdio.h>
#include <strings.h>

int main ()
{
    char str[] ="- This, a sample string.";
    char * pch;
    printf ("Splitting string \"%s\" into
tokens:\n",str);
    pch = strtok (str," ,.-");
    while (pch != NULL)
    {
        printf ("%s\n",pch);
        pch = strtok (NULL, " ,.-");
    }
    return 0;
}
```

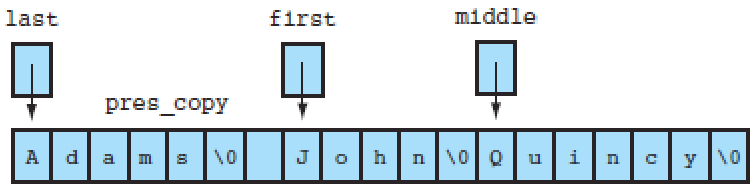
It will produce the following result:

```
Splitting string "- This, a sample string." into tokens:
This
a
sample
string
Program ended with exit code: 0
```



```
char last [20], first  [20], middle  [20];
  char pres[20] = " Adams , John  Quincy ";

strncpy (last, pres, 5);
last[5] = '\0';

strcpy (middle, &pres [12]);

strncpy (first,  &pres[7], 4);
first[4] = '\0';
```

```
char *last, *first, *middle;
char pres[20] = "Adams, John Quincy";
char pres_copy[20];
strcpy(pres_copy, pres);
```



```
last = strtok(pres_copy, ", ");
first = strtok(NULL, ", ");
middle = strtok(NULL, ", ");
```

```
printf("%s\n",last);

printf("%s\n",first);

printf("%s\n",middle);
```

Adams
John
Quincy

Arrays of Strings

An array of strings is in fact a two dimensional array of characters but it is more useful to view this as an array of individual single dimension character arrays or strings.

For Example :-

```
     char str_array[ 10 ] [ 30 ] ;
```

where the row index is used to access the individual row strings and where the column index is the size of each string, thus str_array is an array of 10 strings each with a maximum size of 29 characters leaving one extra for the terminating null character.

For Example :- Program to read strings into str_array and print them out character by character.

```
#include <stdio.h>
char str_array[10][30] ;
int main()
{
    int i, j ;
     puts("Enter ten strings\n") ;
     for ( i = 0 ; i < 10; i++ ) // read in as strings so a
single for loop suffices
     {
        printf( " %d : ", i + 1) ;
        gets( str_array[i] ) ;
     }
     for ( i = 0; i < 10; i++ )//printed out as individual
chars so a nested for loop structure is required
     {
       for ( j=0; str_array[i][j] != '\0' ; j++ )
        putchar ( str_array[i][j] ) ;
       putchar( '\n' ) ;
     }
     return 0;
}
```

Strings and pointers

C's standard library string handling functions use pointers to manipulate the strings. For example the prototype for the strcmp() function found in <string.h> is

```
     int strcmp( const char *string1, const char *string2 ) ;
```

where const is a C keyword which locks the variable it is associated with and prevents any inadvertent changes to it within the function.

Strings can be initialised using pointer or array notation as follows

```
char *str = "Hello\n" ;
char string[] = "Hello\n" ;
```

in both cases the compiler allocates just sufficient storage for both strings.

Example :- Palindrome program using pointers.

```
#include  <stdio.h>
int palin( char * ) ;       /* Function to determine if array is a
palindrome. returns 1 if                             it is a
palindrome, 0 otherwise */
void main( )
{
      char str[30], c ;
      puts( "Enter test string" ) ;
      gets( str ) ;
      if ( palin( str ) )
            printf( "%s is a palindrome\n", str ) ;
      else
            printf( "%s is not a palindrome\n") ;
}
int palin ( char *str )
{
      char *ptr ;
      ptr = str ;
      while ( *ptr  )
            ptr++ ;          /* get length of string  i.e. increment
ptr while *ptr != '\0' */
      ptr-- ;                /* move back one from '\0'             */
      while ( str < ptr )
            if ( *str++  !=  *ptr-- )
                  return 0 ;            /* return value 0 if not a
palindrome */
      return 1 ;             /* otherwise it is a palindrome */
}
```

Arrays of Pointers

It is possible to declare arrays of pointers in C the same as any other 'type'. For example

```
      int *x[10] ;
```

declares an array of ten integer pointers.

To make one of the pointers point to a variable one might do the following.

```
      x[ 2 ]  =  &var ;
```

To access the value pointed to by x[2] we would do the following

```
      *x[ 2 ]
```

which simply de-references the pointer x[2] using the * operator.

Passing this array to a function can be done by treating it the same as a normal array which happens to be an array of elements of type int *.

For Example : -

```
void display( int *q[ ], int size )
{
     int t ;
     for ( t=0; t < size; t++ )
          printf( "%d ", *q[t] ) ;
}
```

Note that q is actually a pointer to an array of pointers as we will see later on with multiple indirection.

A common use of pointer arrays is to hold arrays of strings.

For Example:- A function to print error messages.

```
void serror( int num )
{
     static char *err[] = {
               "Cannot Open File\n",
               "Read Error\n",
               "Write Error\n" } ;
     puts(err[num]);
}
```

Note that using an array of pointers to char initialised as above conserves space as no blank filling characters are required as would be if we used

```
     char err[3][30] = {
          ... } ;
```

5.7 Exercises

1. Write a program that allows the user to read a user specified number of double precision floating point numbers from the keyboard, storing them in an array of maximum size 100 say. Your program should then calculate the sum and the average of the numbers input.

2. Modify your program in exercise 1 so that the maximum and minimum values in the data are found and are ignored when calculating the average value.

3. Write a program that allows the elements of a user input array of doubles to be reversed so that first becomes last etc. Use a separate swap function to carry out the switching of elements.

4. Write a program that reads an array of up to 20 integers from the keyboard and produces a histogram of the values as indicated below.

```
|          *
|        * *
|        * *
|      * * * *
|  *   * * * *   *
|  * * * * * *   *
|* * * * * * * * *
|1 3 2 4 6 7 4 1 3
```

A two dimensional array of characters should be used to produce the histogram, filling it with asterisks appropriate to the data values and then just printing out the array. Some scaling will be required if the values are allowed to exceed the number of asterisks that can fit on the screen vertically.

5. Write a program to accept two strings from the keyboard, compare them and then print out whether or not they are the same.

6(a). Write a function to test whether or not a word is a palindrome e.g. MADAM.

(b). Modify the function in 2(a) so that white space characters are ignored i.e. so that MADAM IM ADAM for example is deemed a palindrome.

7. Write and test a function that inserts a character anywhere in a string. The function should take the general form

strins(char *string, char character, int position)

Notes :-

1. Recall Microsoft C supports a broad range of string handling functions such as strlen(), strcpy(), etc.

2. Your function should provide sufficient error checking to prevent erroneous operation e.g. your function should check that the desired position actually exists.

Modify the strins( ) function written above so that it allows a string of characters rather than an individual character to be inserted at the designated position.

8. Write a program that multiplies two 2 X 2 matrices and prints out the resultant matrix. The program should read in the individual matrices and display them in standard format for the user to check them. The user should be allowed to correct/alter any one element of the matrix at a time and check the revised matrix until satisfied. The results of the operation should be displayed along with the two component matrices.

Recall that $\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} . \begin{pmatrix} c_1 & d_1 \\ c_2 & d_2 \end{pmatrix} = \begin{pmatrix} a_1c_1 + b_1c_2 & a_1d_1 + b_1d_2 \\ a_2c_1 + b_2c_2 & a_2d_1 + b_2d_2 \end{pmatrix}$

Bibliography

problem solving and program design in c - 7th edition (Text Book)
https://www.cs.cmu.edu/~adamchik/15-121/
http://andrei.clubcisco.ro/cursuri/