As per Revised Syllabus of

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

PROGRAMMING IN C & DATA STRUCTURES - *at a Glance*

(One day Revision Book)

Atul P. Godse
M.S. Software Systems (BITS Pilani)
B.E. Industrial Electronics
Formerly Lecturer in Department of Electronics Engg.
Vishwakarma Institute of Technology
Pune

Mrs. Anuradha A. Puntambekar
M.E. (Computer)
Formerly Assistant Professor in
P.E.S. Modern College of Engineering, Pune

TECHNICAL PUBLICATIONS™
An Up-Thrust for Knowledge

technical@vtubooks.com
www.technicalpublications.org

Table of Contents

1 Introduction to Programming

Chapter at a Glance

- When a set of sequential instructions is written to perform any task by a computer, it is called a **'computer program'**.
- Algorithm is a finite sequence of explicit and unambiguous steps required to solve the given problem.

Properties of algorithm

1. The steps used in algorithm must be unambiguous and precisely defined.
2. The uncertainty about the instruction to be executed next, should be avoided.
3. The steps used in algorithm should be finite and the algorithm should be terminated, that is, it cannot be open-ended.
4. The execution of the algorithm should conclude after a finite number of steps.
5. The algorithm must be general enough to deal with any situation.

The key features of an algorithm are :

- **Sequence :** Doing series of actions
- **Decision or selection :** Choosing between two alternative actions
- **Repetition or looping or iteration :** Repeating a series of actions until some condition is or is not present.

- A pictorial representation of an algorithm is called a 'flowchart'.
- Pseudocode is a kind of structured English for describing algorithms. It is made up of two words : **pseudo** and **code**. Pseudo means imitation and code refers to instructions, written in a programming language.

The performance of the algorithms can be measured on the scales of **time** and **space**.

Time : The time it takes for an algorithm to complete. In this case the performance measure is termed **time complexity.** The time complexity of an algorithm or a program is a function of the running time of the algorithm or program.

Space : The memory or non-volatile storage used by the algorithm during its operation.

❑❑❑

2 Introduction to C Language

Chapter at a Glance

Backslash Character Constants (Escape Sequences)

In addition to the character, there can be backslash (\) between the two quote marks. The backslash (\) is known as **escape character**. A backslash followed by one or more special character is known as **escape sequence**. The escape sequences are used to express certain non-printing characters, as well as the backslash and the apostrophe. The Table 2.1 shows the escape sequences used in C.

Escape Sequence	Character Represented
'\0'	**Null character**
'\a'	Alert (bell, alarm)
'\b'	Backspace
'\f'	Form feed (new page)
'\n'	New-line
'\r'	Carriage return
'\t'	Horizontal tab
'\v'	Vertical tab
'\''	Single quotation mark
'\"'	Double quotation mark
'\?'	Question mark
'\\'	Backslash
'\ooo'	Octal number
'\xhh'	Hexadecimal number

Table 2.1 Escape sequences used in C

Basic Data Types and Sizes

C supports several data types, each of which may be represented differently within the computer's memory. The basic data types in C are int, char, float and double. In addition, there are a number of qualifiers that can be applied to these basic types. The qualifiers **short**, **long**, **signed** and **unsigned** apply to integer whereas **long** apply to double. The following tables show the data types with necessary description.

Integer Data Types

Data type	Description	Required Memory in bytes			Range for 16-bit machine	
		16-bit machine	32-bit machine	64-bit machine	Min Value	Max Value
int	signed	2	2	2	– 32,768	32767
	unsigned				0	65,535
short int	signed	2	4	4	– 32,768	32767
	unsigned				0	65,535
long int	signed	4	4	8	– 2,147,483,648	2,147,483,647
	unsigned				0	4,294,967,295

Character Data Types

Data type	Description	Required Memory	Min. Value	Max. Value
char	signed single character	1 Bytes	– 128	127
char	unsigned single character	1 Bytes	0	255

Floating Point Data Types

Data type	Required Memory	Min Value	Max Value
float	4 Bytes	3.4e–38	3.4e+38
double	8 Bytes	1.7e–308	1.7e+308
long double	10 Bytes	3.4e–4932	1.1e+4932

❑❑❑

3 Operators and Expressions

Chapter at a Glance

Arithmetic operators			
Arithmetic operator	Meaning	Declarations : int a = 5, b = 16 double c = 3.0, d = 7.5	
		Integer examples	Floating point examples
+	addition	a + b is 21	c + d is 10.5
–	subtraction	a – b is –11 b – a is 11	c – d is – 4.5 d – c is 4.5
*	multiplication	b * a is 80	c * d is 22.5
/	division	a / b is 0 b / a is 3	c / d is 0.4 d / c is 2.5
%	Remainder (Modulo division)	a % b is 5 b % a is 1	

Fig. 3.1 Arithmetic operations

Relational operators	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
Equality operators	**Meaning**
==	equal to
! =	not equal to

Table 3.2 Relational and equality operators

Logical operator	Meaning
!	NOT
&&	Logical AND
| |	Logical OR

Table 3.3 Logical operators

- Assignment operator assigns the value of expression on the right side of it to the variable on the left of it. The assignment expression evaluates the operand on the right side of the operator (=) and places its value in the variable on the left. Assignment expressions that make use of assignment operator have the form :

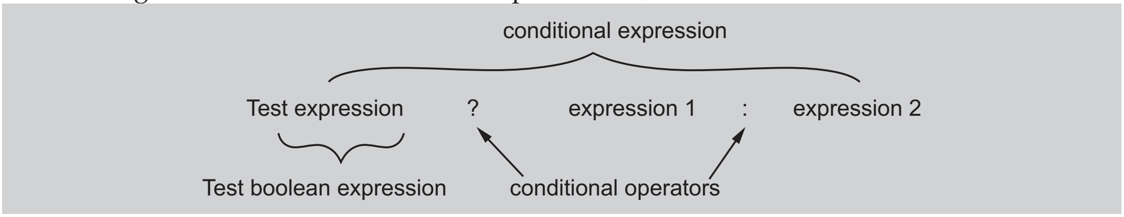
  identifier = expression;

- The operators that act upon a single operand to produce a new value are known as **unary operators**. C support unary operators :

Operators :

  - minus operator –
  - increment operator + +
  - decrement operator – –
  - size operator
  - (type) operator

- In C programming, an operator **sizeof** is used to calculate size of various datatypes.
- The conditional operator (? :) is a **ternary** operator since it takes three operands. The general form of conditional expression is,



- Conditional operator works as follows :
  - The test expression is implicitly converted to boolean expression. It is evaluated.
  - If the result of test expression is **true** (1), the expression 1 is evaluated and the conditional expression takes on value of expression 1.

- If the result of test expression is **false** (0), the expression 2 is evaluated and the conditional expression takes on value of expression 2.

Bitwise Operators

- The bitwise operators are the bit manipulation operators. They can manipulate individual bits within the piece of data. These operators can operate on integers and characters but not on floating point numbers or numbers having double data type. The Table 3.4 lists the various datatypes supported by C.

Operator	Description	Example
~	The bitwise complement operator is an unary operator. It complements each bit of the operand.	Operand = 1111 0000 1111 0000 ~ Operand Resulted operand = 0000 1111 0000 1111
&	The **bitwise-AND operator** compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.	Operand 1 = 1111 1111 1111 0000 Operand 2 : 1111 0000 1111 0000 x = Operand 1 & Operand 2 x = 1111 0000 1111 0000
|	The **bitwise-inclusive-OR** operator compares each bit of its first operand to the corresponding bit of its second operand. If either of bits is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.	Operand 1 : 0000 1111 0011 0000 Operand 2 : 1111 1111 0000 1111 x = Operand 1 | Operand 2 x = 1111 1111 0011 1111
^	The **bitwise-exclusive-OR** operator compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.	Operand 1 : 1111 0000 1100 0011 Operand 2 : 1111 1111 0011 0011 x = Operand 1 ^ Operand 2 x : 0000 1111 1111 0000
<<	The **bitwise shift left operator** is a binary operator. The first operand is the value to be shifted and the second operand specifies the number of bits to be shifted.	Operand 1 : 1111 0000 1111 0001 Operand 2 : 1 X = operand 1 << operand 2 X = 1110 0001 1110 0010 zero is inserted on right

>>	The **bitwise right shift operator** is a binary operator. The first operand is the value to be shifted and the second operand specifies the number of bits to be shifted.	Operand 1 : 1111 0000 1111 0000 Operand 2 : 2 X = operand 1 >> operand 2 X = 00 1111 0000 1111 00 Two zero bits are inserted from left

Table 3.4 Bitwise operators

- Operator **precedence** describes the order in which C evaluates different operators in a complex expression.

Operator type	Operator	Description	Associativity	Precedence level
Parentheses, braces	( )	Parentheses (grouping)	left-to-right	1
	[ ]	Brackets (array subscript)		
Unary	++ – –	Unary preincrement/predecrement	right-to-left	2
	+ –	Unary plus/minus		
	! ~	Unary logical negation/bitwise complement		
	(type)	Unary cast (change *type*)		
	&	Address		
	sizeof	Determine size in bytes		
Binary	* / %	Multiplication / division / modules	left-to-right	3
	+ –	Addition / subtraction	left-to-right	4
	<< >>	Bitwise shift left, Bitwise shift right	left-to-right	5
	< <=	Relational less than / less than or equal to	left-to-right	6
	> >=	Relational greater than/greater than or equal to		
	== !=	Relational is equal to/is not equal to	left-to-right	7
	&	Bitwise AND	left-to-right	8
	^	Bitwise XOR	left-to-right	9
	|	Bitwise OR	left-to-right	10
	&&	Logical AND	left-to-right	11
	||	Logical OR	left-to-right	12
Ternary	? :	Ternary (conditional)	right-to-left	13

Assignment	=	Assignment	right-to-left	14
	+= −=	Addition/subtraction assignment		
	*= /=	Multiplication/division assignment		
	%= &=	Modulus/bitwise AND assignment		
	^= |=	Bitwise exclusive/inclusive OR assignment		
	<<= >>=	Bitwise shift left/right assignment		
Comma	,	Comma(separate expressions)	left-to-right	15

Table 3.5 Operator precedence and associativity

Function	Header file	Description
int **abs**(int x)	stdlib.h	Returns the absolute value of an integer argument x.
double **ceil**(double x);	math.h	Returns the smallest integer value greater than or equal to x.
double **cos**(double x);	math.h	Returns the cosine of a radian angle x.
double **cosh**(double x);	math.h	Returns the hyperbolic cosine of x. The value x is expressed in radians.
double **exp**(double x);	math.h	Returns the value of e raised to the xth power.
double **fabs**(double x);	math.h	Returns the absolute value of x (a negative value becomes positive, positive value is unchanged).
double **floor**(double x);	math.h	Returns the largest integer value less than or equal to x.
double **fmod**(double x, double y);	math.h	Returns the remainder of x divided by y.
int **getchar**();	stdio.h	Returns a character from standard input
double **log10**(double x**) ;**	math.h	Returns the common logarithm (base-10 logarithm) of x.
double **pow**(double x, double y);	math.h	Returns the value of x to the power of y.
printf()	stdio.h	Sends formatted output to standard output.
int **putchar**(int c)	stdio.h	Sends the character given by c to standard output.
int **rand**(void)	stdlib.h	Returns a random positive integer.
scanf()	stdio.h	Reads the formatted input from the standard input.
double **sin**(double x);	math.h	Returns the sine of a radian angle x.
double **sqrt**(double x);	math.h	Returns the square root of x.
void **srand** (unsigned seed)	stdlib.h	Initialises the random number generator

double **tan**(double x);	math.h	Returns the tangent of a radian angle x.
int **toascii**(int c);	ctype.h	Converts value of c to ASCII
int **tolower**(int c);	ctype.h	converts the uppercase letter c to the corresponding lowercase letter.
int **toupper**(int c);	ctype.h	converts the lowercase letter c to the corresponding uppercase letter.

Table 3.6 Commonly used library functions

❑❑❑

4 Input / Output Statements

Chapter at a Glance

Function	Formatted	Unformatted
Input	scanf()	getchar(), gets()
Output	printf()	putchar(), puts()

Table 4.1 I/O functions

- The Fig. 4.1 shows the general form of printf statement.

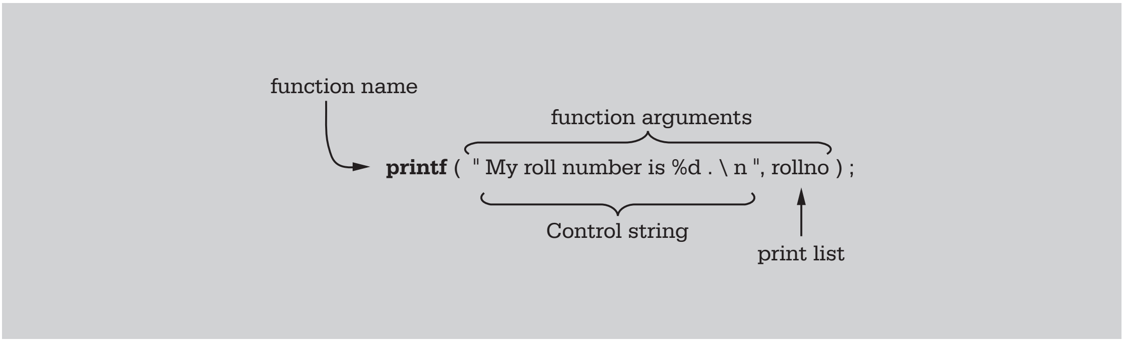


Fig. 4.1 printf statement

- As shown in the Fig. 4.1, the printf statement consists of two parts : the **function name** and the **function arguments**, enclosed in parentheses. The arguments for printf function consists of a **control string** (in quotes) and a **print list** (the variable or variables whose values to be displayed).
- The printf function uses different placeholders to display different type of data items. The Table 4.2 shows the list of placeholders and the type of data item displayed using them.

Placeholder	Type of data item displayed
%c	Single character

%d	Signed decimal integer
%e	Floating point number with an exponent
%f	Floating point number without an exponent
%g	Floating point number either with exponent or without exponent depending on value. Trailing zeros and trailing decimal point will not be displayed.
%i	Signed decimal number.
%o	Octal number, without leading zero
%s	String
%u	Unsigned decimal integer
%x	hexadecimal number, without leading zero.

Table 4.2 Commonly used placeholders in printf function

The putchar Function

- The putchar function displays one character on the display monitor. The character to be displayed is of type **char**. The syntax for putchar function is as given below :

```
putchar (ch_var);
```

Where ch_var is a previously declared character variable.

The puts Function

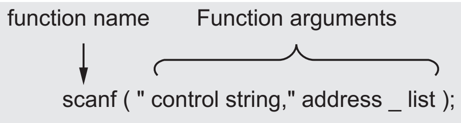
- The puts function displays a string of characters on the display monitor followed by a newline. Its prototype is,

```
puts (string);
```

Where string is a sequence of characters or previously declared array of characters.

The scanf Function

- C provides the scanf function to read data from the standard input device. It is a general purpose input routine. It can read all the build-in data types and automatically convert numbers into the proper internal format. It is much like a reverse of printf function. The prototype for scanf function is,



```
scanf ( " control string," address _ list );
```

Fig. 4.2

- The arguments for scanf function consists of a control string (in quotes) and an address_list address/es of memory location/s where the values of input variable/s should be stored. The control string determines how values are read into the variables.
- Like printf function, the scanf function uses format specifiers preceded by a % sign to tell scanf function what type of data is to be read. The Table 4.3 shows the list of format specifiers and the type of the data can be read using them.

Format specifier	Type of data item read
%c	Single character
%d	Signed decimal integer
%e	Floating point number
%f	Floating point number
%g	Floating point number
%h	Short integer
%i	Integer in either decimal, octal or hexadecimal format.
%o	Octal integer
%s	Character string
%u	Unsigned decimal integer
%x	Hexadecimal integer
%[ ]	Set of characters
%%	Percent sign

Table 4.3 Commonly used format specifiers in scanf function

The getchar Function

- The getchar function accepts a single character from the keyboard. The function does not require any arguments, though a pair of empty parentheses must follow the word getchar as a syntax. It returns a single character from a standard input device (typically a keyboard) and it can be assigned to predeclared character variable.

```
ch_var = get char( );
```

The gets Function

- The gets function reads a string of characters from the keyboard and stores them at the address pointed by its argument. It reads the string of characters until the Enter or Return key is pressed. The Enter key or Return key does not become part of the string; instead, a null terminator is placed at the end. We can correct typing mistakes by using the backspace key before pressing Enter key. The prototype for gets() is

```
gets(str);
```

- Where str is name/pointer to character array that receives the characters entered by the user.

❑❑❑

5 Branching and Looping

Chapter at a Glance

The if-else Statement

- The **if** statement is the simplest decision making statement. It is used to decide whether to do something at a special point, or to decide between two courses of action.

The if-else Statement Syntax

```
if   (expression) ◌  - No Semicolon
     statement1;      } Single statement if body
else
     statement2;      } Single statement else body
```

where the **else part** is **optional**. The test expression within the brackets following the **if** is evaluated; if it is true (that is, if expression has a non-zero value), statement1 is executed. If it is false (that is expression has a zero value) and if there is an else part, statement2 is executed instead.

```
if   (expression) ◌  - No Semicolon
     {
         statement;
                         } Multiple statements if body
         statement;
     }
else
     {
       statement;
                         } Multiple statements else body
       statement;
     }
```

Nested if Statement

- In the if-else statement, if body of either if or else or both include another if-else statement, the statement is known as **nested if**. Nested ifs are very common in programming. They are used when there are two or more alternatives to select.

Nested if Syntax

```
if (expression 1)          - No semicolon
{
    if (expression 2)      - No semicolon
        statement;
    else
        statement;
}
else
    statement;
```

The if-else-if Ladder

- Sometimes we wish to make a multi-way decision based on several conditions. The most general way of doing this is by using the else if variant on the if statement. This works by cascading several comparisons. As soon as one of these gives a true result, the following statement or block is executed, and no further comparisons are performed. If none of the conditions is true, the final **else** is executed. The final **else** statement is optional. If final **else** is not present, no action takes place if all other conditions are false.

The else-if Statement Syntax

```
if   (expression)
     statement ;
else if (expression)
     statement ;
else if (expression)
     statement ;
else if (expression)
     statement ;
else
     statement ;
```

Switch Statement

- The **switch** statement is a multi-way decision that tests whether an expression matches one of a number of **constant integer** values, and branches accordingly. This is another form of the multi way decision. It is well structured, but can only be used in certain cases where :
  - Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type (int, long, short or char).
  - Each possible value of the variable can control a single branch. Default branch may optionally be used to trap all unspecified cases.

The switch Statement Syntax

```
                         integer or character variable
                         or an expression

switch (expression)          - No semicolon
{
case  constant1:
              statement;      } constant 1 body
              statement;
              break ;               -------------------- causes exit from switch
case  constant2 :
              statement;      } constant 2 body
              statement;
              break;
              :

default :
              statement;      } Default body
              statement;
}      - No semicolon
```

Important Points

- The **switch** differs from the **if** in that switch can only test for equality, whereas if can evaluate any type of relational or logical expression.
- No two **case** constants in the same **switch** can have identical values. However, a **switch** statement enclosed by an outer switch (i.e. nested **switch**) may have **case** constants that are in common.
- If character constants are used in the **switch** statement, they are automatically converted to integers.
- The switch is often used to process keyboard commands, such as menu selection.
- In switch statement, the case labels that contain type **double** values or **strings** are not permitted.

The goto Statement and Labels

- The **goto** statement is used to alter *unconditionally* the normal sequence of program execution by transferring control to some other part of the program.

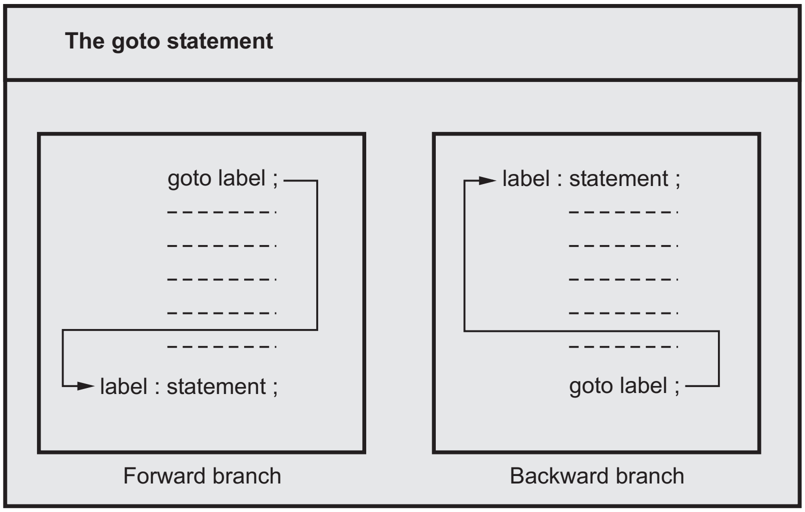


Fig. 5.1

- Depending on the position of the control statement in the loop, the loop statement is classified as **pre-tested loop** (also known as **entry-controlled loop**) or **post-tested loop** (also known as **exit-controlled loop**).
- In **pre-tested loop**, the control statement performs a logical test before the execution of the loop. If the test is true, the loop is executed; otherwise it is terminated.

- In **post-tested loop**, the body of the loop is executed unconditionally for the first time because a logical test is performed after the execution of the loop body. If the test is true, the new iteration of loop is executed; otherwise loop is terminated.

The Fig. 5.2 illustrates the operation of pre-tested and post-tested loops

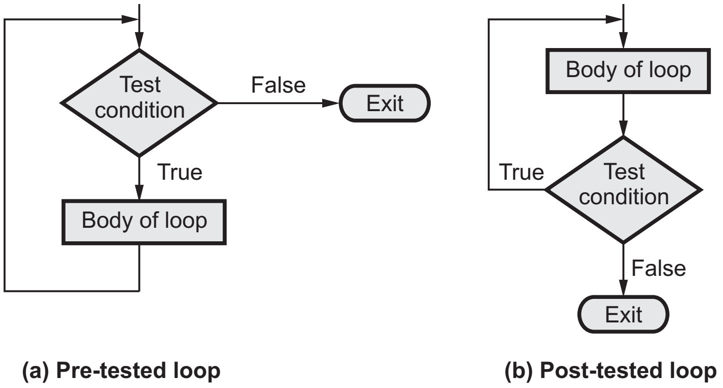


Fig. 5.2 Operation of pre-tested and post-tested loops

There are three loop control elements. These are :

- **Initialization expression :** It sets initial value for the loop counter before the statements within the loop body start to execute.
- **Test expression :** It tests the condition to determine whether to execute the loop or not.
- **Update expression** : It increments or decrements the loop counter value such that after appropriate iterations, test condition will result false and loop will be terminated.

C gives us a choice of three types of loop : while, do while and for.

- The **for loop** is frequently used, usually where the loop is to be executed a fixed number of times.
- The **while loop** keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop to be executed.
- The **do while loop** is similar, but the test occurs after the loop body is executed. This ensures that the loop body is run at least once.
- In **for loop**, initialization, condition and adjustment statements are all put together in one line which make loop easier to understand and implement. While in the **while loop**, initialization is done prior to the beginning of the loop and conditional statement is always put at the start of the loop. On other hand, in **do ... while loop**, initialization is done prior to the beginning of the loop and conditional statement is always put at the end of the loop.

The for loop Syntax

```
for (initialization expression; test expression; update expression)     - No semicolon
     statement ;      }  Single statement loop body
Note  : Curly braces are optional if loop body consists of single statement
for (initialization expression; test expression; update expression)     -No semicolon
{
     statement;  )
     statement;  }     Multiple statement loop body
     statement;  )
}
```

Nested for Loops

- Like **if** statement, for statement can be nested. The one **for** statement within another **for** statement is called **nested for loops**. This is illustrated in Fig. 5.3.

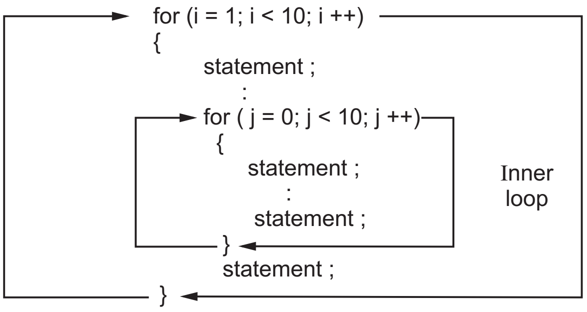


Fig. 5.3 Nested for loops

The break Statement

- The break statement has two uses. We have seen the use of it to terminate a **case** in the **switch** statement. We can also use break statement to force immediate termination of loop, bypassing the normal loop conditional test. When **break** is encountered inside any loop, control automatically passes to the first statement after the loop.

The while Statement

- The while loop is a pre-test loop. Since it is a pre-test loop, it tests the expression before every iteration of the loop. It repeats a loop body until the logical test at the top proves false.
- The while loop looks like a simplified version of the **for** loop. It contains a test expression but no initialization or update expressions. Although there is no initialization expression, the loop variable

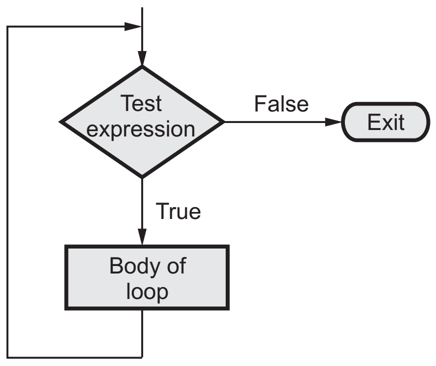


Fig. 5.4 Operation of the while loop

must be initialized before the loop begins. The loop body must also contain some statement that updates the value of the loop variable; otherwise the loop would never end.

- The Fig. 5.4 shows the operation of while loop.

The while loop Syntax

```
while (test condition)          - No semicolon
    statement ;       Body of the loop with single statement
Note : Curly braces are optional if loop body consists of single statement

while (test condition)          - No semicolon
{
    statement ;
    statement ;       Body of the loop with multiple statements
}
```

The do while Statement

The **do while** Loop is very similar to the **while** loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before terminating.

The do while loop Syntax

```
do          - No semicolon
statement ;                 Body of the loop with single statement
while (test condition) ;

Note : Curly braces are optional if loop body consists of single statement

do          - No semicolon
{
    statement ;
    statement ;       Body of the loop with multiple statements
} while (test condition) ;
```

- The Fig. 5.5 shows the operation of the **do while** loop.
- The **do while** loop is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

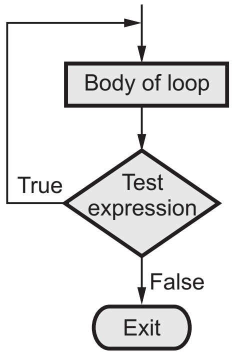


Fig. 5.5 Operation of the do while loop

When to use which loop

- So far we have seen how **for, while** and **do while** loops are executed. Let us see when we should use which loop. The **for** loop is appropriate when we know in advance how many times the loop will be executed. The **while** and **do while** loops are used when we don't know in advance when the loop will terminate. The **while** loop should be used when we want to execute the loop body even once, and the **do while** loop when we want to execute loop body at least once.

Difference between while and do..while loop

No.	While Loop	Do-While Loop

1.	**Syntax :**
	while (Condition)
	{
	statements;
	}
		Syntax :
		do
		{
		statements;
		} while (Condition);

2.	No semicolon after condition in the syntax.	There is semicolon after condition in the syntax.
3.	Entry Controlled loop.	Exit Controlled loop.
4.	If the condition is not true loop is never executed.	Loop is executed at least once even though condition is not true.

The continue Statement

- The **continue** statement only works within loops. It forces the next iteration of the loop to take place, skipping the remaining unexecuted statements in the loop.

- In the **for** loop, **continue** causes the increment and then the condition test portions of the loop to execute.
- In a **while** and **do while** loops, **continue** passes the program control to the conditional tests.

- Like **break** statement, the continue statement is usually associated with an **if**. The Fig. 5.6 shows the operation of continue statement.

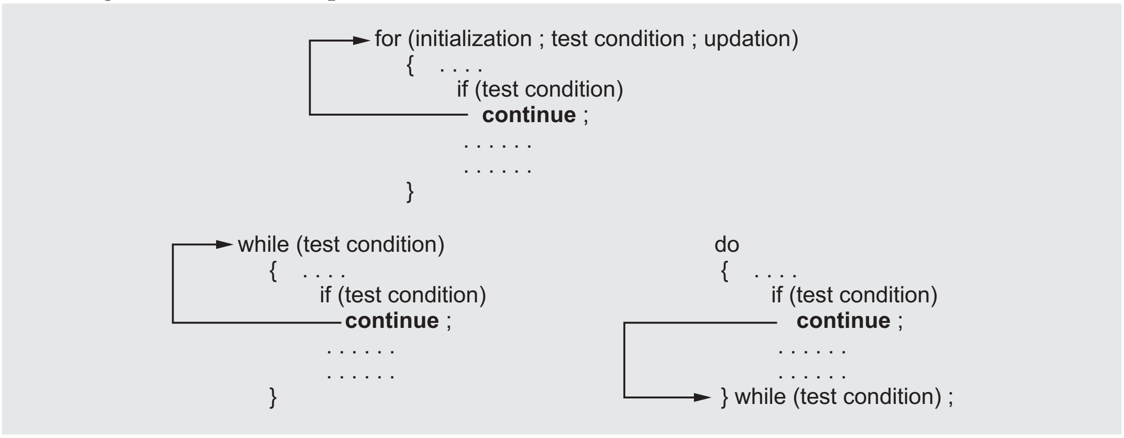


Fig. 5.6 Operation of continue statement

❑❑❑

Notes

6 Arrays and Strings

Chapter at a Glance

- Array can be defined as a collection of variables of the same type that are referred through a common name. A specific element in an array is accessed by an index. In C, all arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

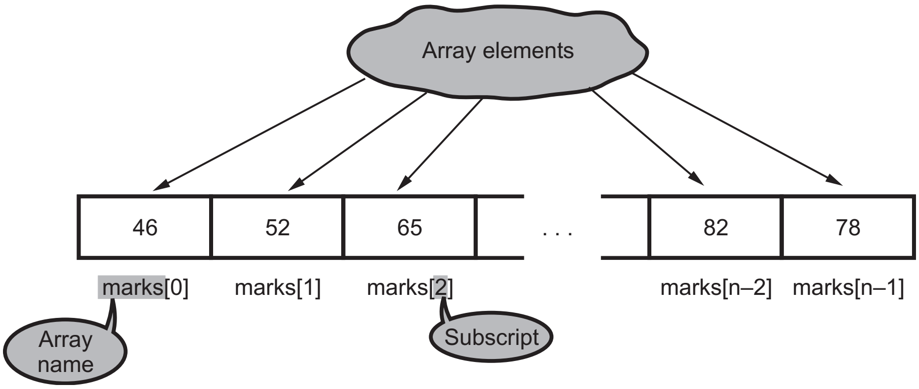


Fig. 6.1 Structure of an array named mark having n elements

Array Declaration and Definition

- Like other variables, an array needs to be declared and defined at the beginning so that the computer will know what is the data type of array elements and how large the array size, i.e., the maximum number of elements that array can hold. The general form for declaring an array is

```
storage_class type var_name[size];
```

- where storage_class refers to the storage class of the array. In our program, we have declared a 50-element array called **marks** of type **float** using statement :

```
        float marks [50];
```

- Here, **float** specifies the type of the variable the word marks specifies, the name of the array. The [50] tells, there are maximum 50 numbers of type float in the array. Note the storage class is not specified; by default it is **auto**.

- Usually, the maximum size of an array is defined by the symbolic constant instead of fixed number, and we can use the same symbolic constant to represent the maximum size of the array in the program when it is necessary to process array elements. With such a definition, if we want to change the size of the array, it can be easily done by modifying the definition of the symbolic constant and it is not necessary to change the code where maximum size of an array is specified.
- The amount of storage required to hold an array is directly related to its type and size. Therefore, the total size of an array in bytes can be computed as :

  Total bytes = Size of (base type) × Length of array i.e., number of array elements

In our case, Total bytes = Size of (**float**) × 50

Storing Elements in Array

- An array must be both, declared and defined before it can be used. Array declaration only reserves memory for the elements in the array. No values will be stored. The values of array elements are stored when array is defined. This is also called **array initialization**.
- Like variables, the array elements can be initialized when array is defined or they can be initialized within the program. Let us see how they can be initialized at the time of declaration. The general form of initialization of array is :

```
type array_name [size] = {value 0, value 1, .........., value n – 1};
```

- The values are specified in the curly brace and they are separated by commas.

Important Points

- Array is a collection of variables of same type.
- Subscript in square brackets is used to indicate particular element in the array.
- Subscript can be integer variable or integer number

  i = 5; a = marks[i]; and a = marks [5];

  give the same result.
- The first element in the array is numbered 0, so the last element is 1 less than the size of array.
- Array of characters is known as string and it is ended with null character('\0').
- An array elements are always stored in contiguous memory locations.

Important Points - Multi-dimensional array definition

- The number of inner pairs which stores sets of values must be equal to the number of rows in the array.
- The first set of values are assigned to the first row elements, the second set of values are assigned to the second row elements and so on.
- In the definition of the array if all values are not specified, the specified values are assigned to corresponding array element and remaining array elements are initialized to zero value.
- If the number of values in each inner pair of braces exceeds the defined array size, the compilation error will occur.

Strings

- A string is a sequence of characters that is treated as a single data item. It can be represented as a one-dimensional character-type array and each character within the string represents one array element.

Reading Strings

- We can use any one of the following three functions to read strings.

  1. scanf 2. getchar 3. gets

Writing Strings

- Strings can be displayed on the screen using any of the following functions.

  1. printf 2. putchar 3. puts

❑❑❑

Notes

7 Functions

Chapter at a Glance

- In C, the idea of top-down design is implemented using **functions.** A function in C is an independent module that will be called to do a specific task. A **called function** receives the control from a **calling function**. When the called function completes its task, it returns to the calling function.

General Form of a Function

The function definition contains the function_name with parameter list and the type specification of the value return by the function (optional) and the function body.

Function Definition Syntax- non ANSI Function

$$\begin{array}{l}\textbf{return\_data\_type}\ \text{function\_name}\ (\ \textbf{parameters list}\ )\\ \quad \text{data\_type\_declarations\_of\_parameters;}\\ \quad \{\\ \quad\quad \left.\begin{array}{l}\text{variable declarations;}\\ \text{statement1;}\\ \text{statement2;}\\ \quad .\\ \quad .\\ \text{return(value)}\end{array}\right\}\ \text{function\_body}\\ \quad \}\end{array}$$

The description of each component of function definition is as given below.

- The return_data_type represents the type of data we want to return upon completion of the function. Any valid datatype can be used.
- The function_name is the identifier name of the function. It follows the same variable naming convention.

- The parameter list defines the parameters/arguments that must be specified when the function is called. The function can accept an unspecified number of parameters/arguments.
- ANSI C varies slightly in the way that functions are declared. Its format is

Function Definition Syntax - ANSI Function

```
return_data_type function_name ( parameters declarations )
      {
      variable declarations;      \
      statement1;                 |
      statement2;                 |
        .                         }  function_body
        .                         |
      return(value)               /
      }
```

Declaring Function / Function Prototype

- Like variables a function must be declared before it can be used (called). The function declaration includes the return type and the number and type of the arguments. This is also called the **function prototype**. The declaration of a function resembles the definition,

Function Declaration

return-type function-name (**parameter types**);

Example : declaring average

double average (**double, double**);

- In declaration, the function body is replaced by a semi-colon. The function parameters declared in the header of the function declaration is called **Formal parameters**. This list defines and declares the variables that will contain the data received by the function. Formal parameters need not be named, it is sufficient to specify their types.

Important Points

- A function can be called from another function.
- A function can be called within itself. This is called a **recursive** function.
- A function can be called any number of times.
- When a function is called, it has control of that process until it returns.

Functions and Arguments (Parameters)

- The **calling** function conveys the information to the **called** function using the mechanism called **arguments**. We have unknowingly used the arguments in the **printf**() and **scanf**() functions; the format string and the list of variables inside the parenthesis in these functions are arguments. The arguments are also called **parameters.** There are two methods of passing arguments/parameters to a function.
  - Call by value
  - Call by reference

Call by Value

- When a function passes values to the called function, the values of the arguments are copied to the corresponding parameters. This ensures that the changes made to the parameters inside the function will not affect the argument (the value of variable in the calling function).

Call by Reference

- It is the second way of passing arguments to the function. In this method, the **address** of an argument is copied into the parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Return Statements

- The **return** statement causes an immediate exit from the function. That is, it causes program execution to return to the calling code. The return statement also can be used to return a value.

Important points – Recursive function

- Recursive function uses **stack**. A stack is a Last In First Out (LIFO) data structure. This means that the last data item to get stored on the stack (often called push operation) is the first data item to get out of the stack (often called pop operation). This is similar to that of stacking plates - the last plate that goes on the stack is the first one to get out of it.
- When function calls itself, a new set of variables and parameters are stored on the stack and the function code is executed from the top with these new variables.
- As each recursive call returns, the old variables and parameters declared in the function are removed from the stack, and execution resumes immediately after the recursive call inside the function.
- The recursive function must have a conditional statement, such as **if**, somewhere to force the function to return without the recursive call being executed.

❑❑❑

8 Structures

Chapter at a Glance

Defining a Structure

- In C, a **structure** is a derived data type consisting of a collection of member elements and their data types. Thus, a variable of a structure type is the name of a group of one or more **members** which may or may not be of the same data type. In programming terminology, a structure data type is referred to as a **record data type** and the members are called **fields.**
- For defining the structures, we must specify the names and types of each of the fields of the structure and declare variables of that type. Here, is an example structure definition.

```
struct student
            {
               char name[64];
               char course[128];
               int age;
               int year;
            } ;
```

- This defines a new data type student. It contains four members :
  - 64 - element character array, **name [64]**.
  - 128 - element character array, **course [128]**.
  - Integer quantity, **age**.
  - Integer quantity, **year**.
- The composition of this student structure is illustrated in Fig. 8.1.

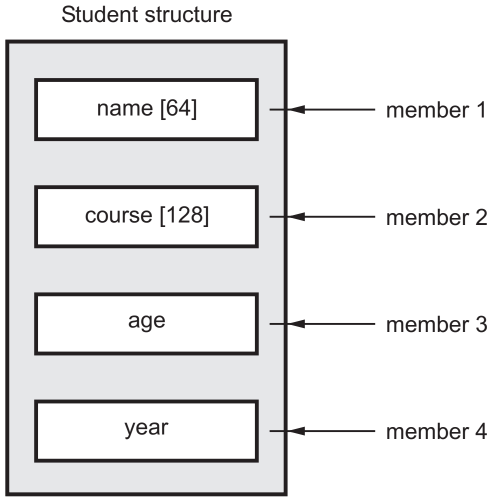


Fig. 8.1 Structure student

- In general terms, the composition of structure may be defined as

```
struct tag
        {
          member 1;
          member 2;
              :
              :
          member n;
        };
```

where struct is a keyword.

tag is a name that identifies structures of this type.

- The variables of type student can be declared as follows.

```
struct student s1, s2 ,s3;
```

- The s1, s2, s3 are variables of data type student. They have members(fields) called name, course, age and year. In general terms, individual structure-type variables can be declared as :

```
struct tag variable 1, variable 2, variable 3, . . . ,variable n;
```

Declaring structure variable includes :

- The keyword struct
- The structure tag name
- List of variables separated by commas
- A terminating semicolon

Accessing and Processing a Structure

- The members of a structure can be accessed and processed as separate entities. A structure members can be accessed by using dot (.), also called **period** operator. The syntax is,

```
variable . member
```

- where **variable** refers to the name of a structure type variable, and **member** refers to the name of a member within the structure.

Structures compared with arrays

- Like arrays, structures are single, named items which on closer examination have internal fields which can hold values.
- Array is a derived data-type whereas structure is a programmer defined data-type.
- Unlike arrays, the members can be of different types and lengths.
- Unlike arrays, for structs the order of the members is irrelevant. They are to be thought of as simply grouped together for programming convenience.
- We can only access structure members by naming them, not by indexing.
- Like arrays we can use the **&** operator to get the address of the start of the structure and this is often a convenient way of passing structures to and from functions.

- The Fig. 8.2 shows the array of structure student. It can store information of 40 students in the class.

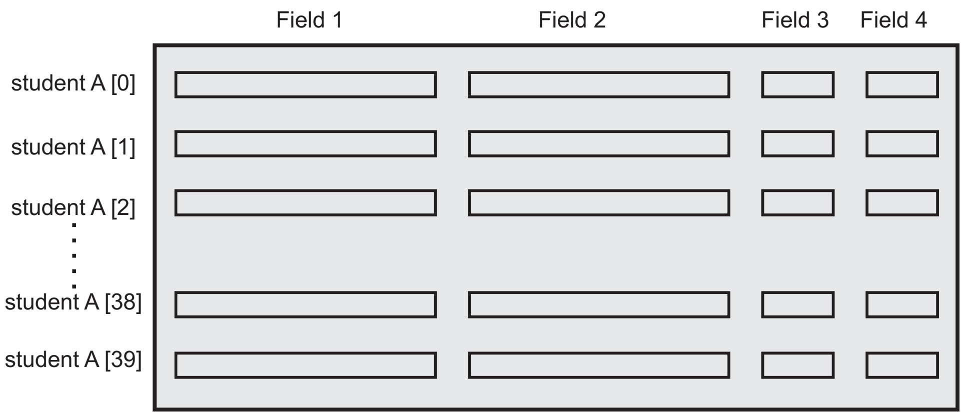


Fig. 8.2 Array of structure

❑❑❑

9 File Management

Chapter at a Glance

- The primary difference between manipulating files and doing terminal I/O is that we must specify in our programs which files we wish to use. As we know, we can have many files on our disk. If we wish to use a file in our programs, then we must specify which file or files we wish to use. Specifying the file we wish to use is referred to as **opening** the file.
- When we open a file, we must also specify what we wish to do with it i.e. **Read** from the file, **Write** to the file, or both. Because we may use a number of different files in our program. We must specify **when** reading or writing which file we wish to use. C communicates with files using a new datatype called a **file pointer**.
- Every file we open has its own **file pointer variable**. When we wish to write to a file, we specify the file by using its file pointer variable. We can declare these file pointer variables as follows :

```
FILE  *fp1, *fp2, *fp3;
```

- The variables fp1, fp2, fp3 are file pointers.

Important File Handling Functions Supported by C

The Table 9.1 lists the important file handling functions supported by C.

Function	Operation
fopen()	Opens an existing file or if not existing creates a new file for use
fclose()	Closes a file
putc ()/fputc()	Writes a character to a file
getc ()/fgetc()	Reads a character from a file
fprintf()	Same as printf() except that it writes data values in the file instead of displaying on the screen.

fscanf()	Same as scanf () except that it reads set of data values from a file instead of keyboard.
getw()	Reads an integer from a file.
putw()	Writes an integer to a file
fputs()	Writes a string to a file
fgets()	Reads a string from file
fseek()	Sets the file position indicator to a desired point in the file.
ftell()	Returns the current file position.
rewind()	Resets the file position indicator to the beginning of the file.
feof()	Returns true if end-of-file is reached.
ferror()	Returns true if an error has occurred.
remove()	Erases the file.

Table 9.1 Commonly used C file functions

- The general format of **fopen** function is

```
File_pointer_name = fopen (file_name, mode)
```

- The **fopen** function has two parameters; the name of the file to be opened and the mode to use. It returns a pointer to a FILE which it creates.

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+	Append or create a text file for read/write

Table 9.2 File modes for text files

- The function **fclose** is used to **close** the file i.e. indicates that we have finished processing this file. Systems have a limit on the number of files which can be opened simultaneously, so it is a good idea to close a file when we have finished using it. The prototype is

```
fclose (File_pointer_name);
```

- The argument for **fclose** function is the file pointer. For example,

```
fclose (fp);
```

- This statement closes the file associated with **fp**, which must be a valid **file pointer** previously obtained using **fopen().** The **fclose()** function returns 0 if successful and **EOF** (end of file) if an error occurs.
- We can use **fgets** and **fputs** functions to read string from and write string into file, respectively. For formatted input or output of files, we can use functions **fscanf** and **fprintf**, respectively.

❑❑❑

Notes

10 Pointers

Chapter at a Glance

- A pointer is a variable that represents the location (rather than the value) of a data item, such as a variable or an array element. It is a variable that holds a memory address. This address is the location of another variable or an array element in memory. For example, if one variable contains the address of another variable, the first variable is said to **point** to the second. This is illustrated in Fig. 10.1.

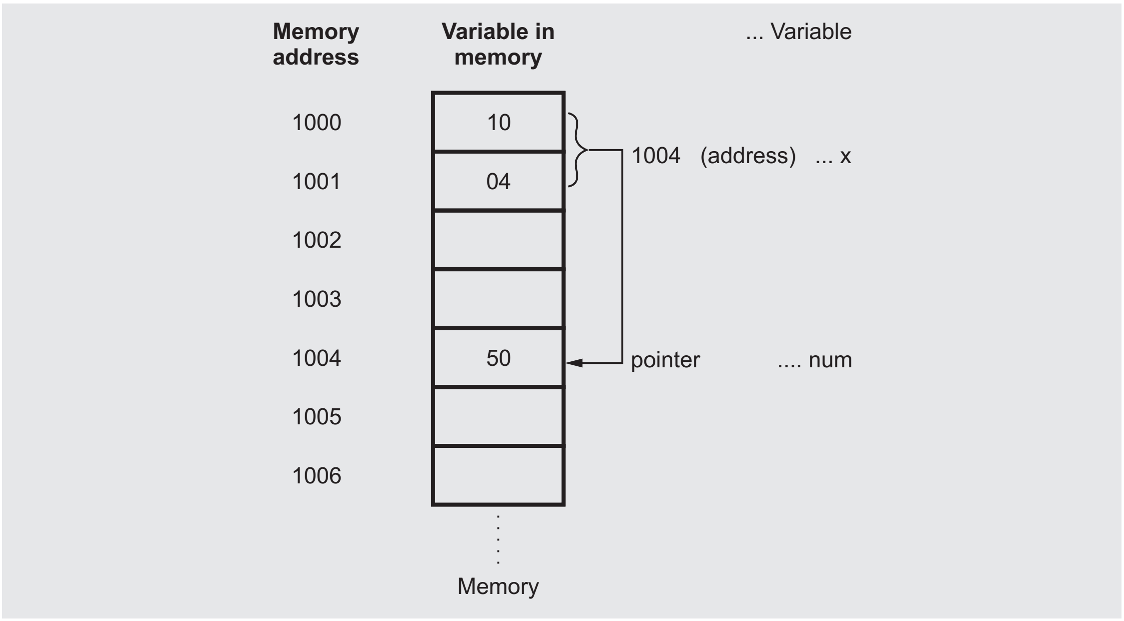


Fig. 10.1 One variable (pointer) points to another

Accessing the Address of a Variable

- The actual location (address) of a variable in the memory is not known to the programmer directly. However, programmer can determine the location of a variable in the memory using **address operator '&'**. For example,

```
x = & num;
```

assigns the address 1004 (the location of num) to the variable x.

Declaring Pointer Variables

- If a variable is going to be a pointer, it must be declared as such. A pointer declaration consists of a base type, an * and the variable name. The general form for declaring a pointer variable is

```
    data_type  * var_name;
```

- For example, consider the variable declaration :

```
    int *ptr;
```

- ptr is the name of our variable. The '*' informs the compiler that we want a pointer variable, i.e. to set aside number of bytes required to store an address in memory. The **int** says that we intend to use our pointe pointer is said to "point to" an integer. However, note that when we wrote **int k;** we did not give **k** a value. Similarly, **ptr** has no value, this means that we haven't stored an address in it in the above declaration.

Initializing Pointer Variables

- The process of assigning the address of a variable to a pointer variable is known as **initialization of pointer variable**. We can store the address of our integer variable **k** in **ptr** (pointer variable). To do this we use the unary **&** operator and write :

```
ptr = &k;
```

- The **&** operator retrieves the address of **k**, even though **k** is on the right hand side of the assignment operator '=', and copies that to the contents of our pointer ptr.

Accessing a Variable Through its Pointer

- We have seen how to store address in the pointer variable. We can also store the value at the address specified by the pointer variable. For this, we have to use the **dereferencing operator** (asterisk) and it is used as follows :

```
int k, *ptr;
ptr = &k;
*ptr = 7;
```

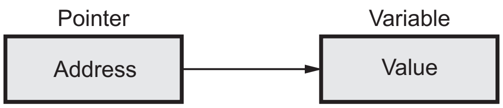
- The statement *ptr = 7 will copy value 7 to the address pointed to by **ptr**. The **ptr** "points to" (contains the address of) **k**, therefore, the above statement will set the value of **k** to 7. That is, when we use the '*' this way we are referring to the value of that which ptr is pointing to, not the value of the pointer itself.

Important points - Pointer

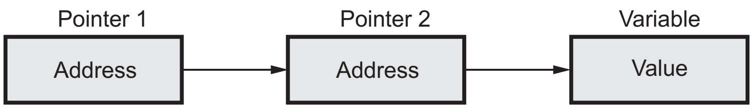
- A pointer variable can be assigned the **address** of an ordinary variable. For example, p = &k where p is pointer variable and k is ordinary variable of **same data type** of that of pointer variable.
- A pointer variable can be assigned the **value** of another pointer variable. For example, p = q where p and q are both pointer variables of **same data type**.
- A pointer variable can be assigned a NULL (zero). For example, p = NULL.
- An integer quantity can be added to or subtracted from a pointer variable. For example, p+3, p++, – – p where p is a pointer variable.
- One pointer variable can be subtracted from another pointer variable provided both pointers point to elements of the same array.
- Two pointer variables can be compared provided both pointers point to objects of the same data type.

Pointers to Pointer

- We can have a pointer to point to another pointer that will point to the target value. This situation is called **multiple indirection,** or **pointer to pointers.**



a) Single indirection



b) Multiple indirection

Fig. 10.2 Differentiating single and multiple indirection

Pointer to Pointer Declaration

- A pointer to pointer variable is declared by placing an additional asterisk in front of the variable name as shown below :

```
float ** amount ;
```

- Here, amount is a pointer to pointer of type **float.**

❑❑❑

Notes

11 Preprocessors

Chapter at a Glance

- The C compiler is made of two functional parts : a **preprocessor** and a **translator**. The preprocessor uses programmer-supplied commands to prepare the source program for compilation. The translator then accepts source program and converts it into machine code and generates the object module.

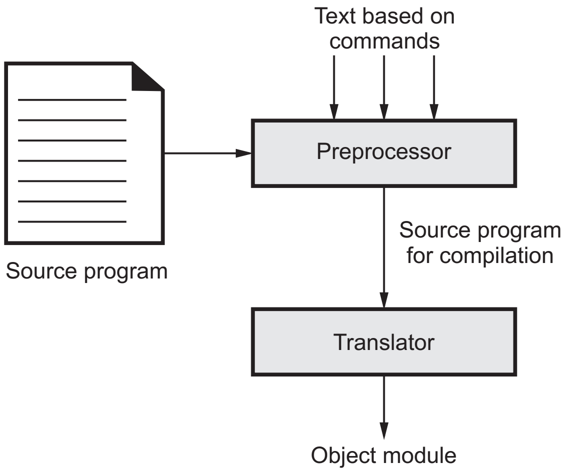


Fig. 11.1 Compiler environment

- The preprocessor inserts, includes, excludes and replaces text based on commands supplied by the programmer. All preprocessor commands start with a pound sign (#). The preprocessor performs three different functions.
  - It is used to specify which header files are to be included in the program, e.g. #include.
  - It is used for macro definition, which enables us to create macros for use in our program, e.g. #define.
  - It can be used to conditionally include or exclude code from the program.

File Inclusion

- The preprocessor **#include** is used to copy the contents of other files into program. The files are usually header files that contain prototype statements and data declarations for the program.

Macro Substitution

- The second preprocessor function is macro definition. A macro is formal syntax that can be used to generate statements for use in a program. For the C language, the macro generates C statements. There are two types and macro formats : **simple** and **parameterized**.

Compiler Controlled Directives

- The third use of the preprocessor is to control the selective compilation of portions of the program code. This is useful in inserting debugging logic in a program and commenting out code. The Table 11.1 shows various conditional code commands supported by C.

Command	Description
#if	If expression : when true following code included.
#endif	End of if expression : terminates included statements.
#else	Specifies alternate code to be included when if expression results false.
#elif	else-if : specifies alternate code to be included when previous conditional statement results false.
#ifdef	if defined : include following statements when a macro name is defined by #define.
#ifndef	if not defined : include the following statements when a macro name is not defined.

Table 11.1 Conditional compilation commands

❑❑❑

12 Introduction to Data Structures

Chapter at a Glance

Concept of Data Structure

- The data structure can be defined as the **collection of elements** and all the **possible operations** which are required for those set of elements. In other words data structure will tell us the elements that are required as well as the legal operations on those set of elements.

Example of Data Structure

- Consider a set of elements which are required to store in an array. Various operations such as reading of the elements and storing them at appropriate index can be performed. If we want to access any particular element then that element can be retrieved from the array. Thus reading, printing, searching would be the operations required to perform these tasks for the elements. Thus data object integer elements and set of operations form the data structure-array.

Formal Definition of Data Structure

- A data structure is a set of domains D, a set of functions F and set of axioms A. This triple (D, F, A) denotes the data structure d.
- **Data type** is a type of data. Typically data type is specified at the time of declaring the variable.

  For example :

  ```
  int a,b,c;
  ```

- Here **int** is a data type that represents the numerical value and a,b, and c are the identifiers or variables.
- In C there are four basic data types. Sometimes these data types are also called as **primitive data types.**
- There two types of data types - 1. Primitive Data types and 2. User Defined Data types.

Primitive data types :

1. Integer Data Type :

- This data type is used to have the integer value in a variable. The value should not contain any fractional data.

2. Float Data Type :

- This data type is used to store the real values in a variable. The value may contain some fractional data.

3. Character Data Type :

- This data type is used to have some text or alphabetical information in a variable.

4. Double Data Type :

- This data type is used to store the numeric information in the variable. This data type is mainly meant for the real values. It is same as of float data type but the capacity of this data type is larger than the float data type.

Types of Data Structure

- The data structures can be divided into two basic types preliminary data structure and secondary data structure. The Fig. 12.1 shows various types of data structures.

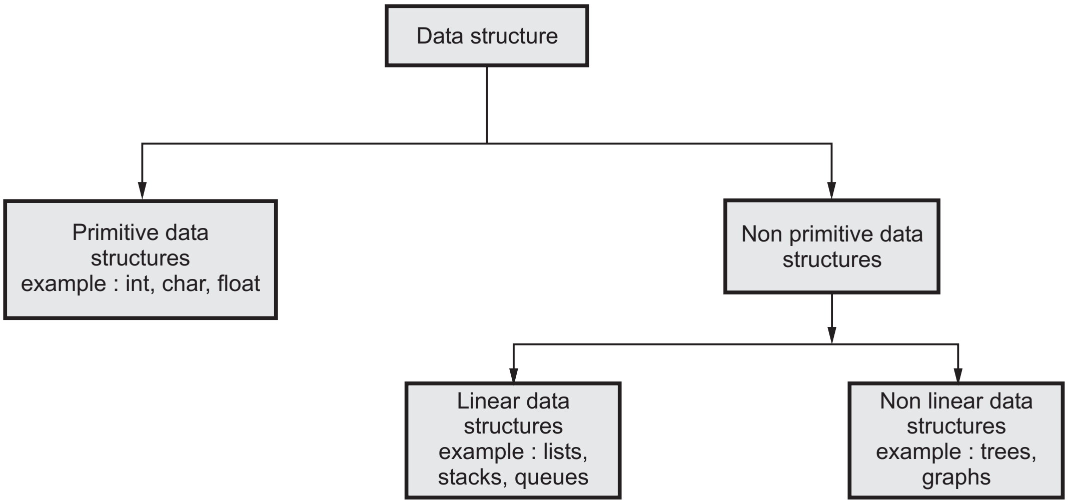

Data structure
Primitive data structures example : int, char, float
Non primitive data structures
Linear data structures example : lists, stacks, queues
Non linear data structures example : trees, graphs

Fig. 12.1 Classification of data structure

Abstract Data Types

- The abstract data type is a triple of D - set of domains, F - set of functions, A - axioms in which only what is to be done is mentioned but how is to be done is not mentioned.

Stack

A stack is an ordered list in which all insertions and deletions are made at one end, called the top. If we have to make stack of elements 10, 20, 30, 40, 50, 60 then 10 will be the bottommost element and 60 will be the topmost element in the stack. A stack is shown in Fig. 12.2.

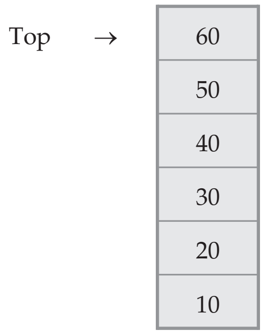


Fig. 12.2 Stack

- Various operations on stack are -
  1. Creation of stack
  2. Insertion of element onto the stack
  3. Deletion or popping of element from the stack

The 'Push' and 'Pop' Functions

```
void push(int item)
{
 st.top++; / * top pointer is set to next location */
 st.s[st.top] =item; /* placing the element at that location */
}
```

```
int pop()
{
 int item;
 item = st.s[st.top];
 st.top – –;
 return(item);
}
```

Queue

- The queue can be formally defined as ordered collection of elements that has two ends named as **front** and **rear**. From the front end one can delete the elements and from the rear end one can insert the elements.

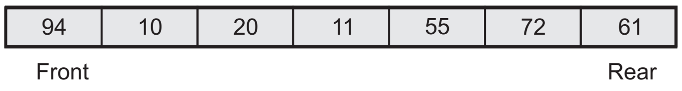


Fig. 12.3 Queue

- Various operations on the queue are -
  1. Queue overflow.
  2. Insertion of the element into the queue.
  3. Queue underflow.

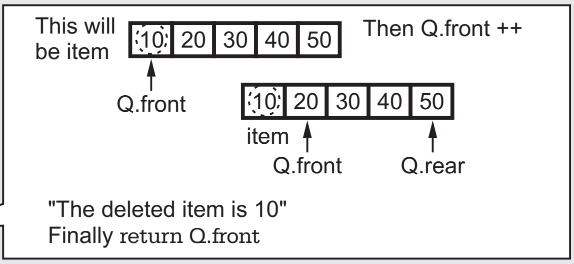
4. Deletion of the element from the queue.
5. Display of the queue.

```
int insert(int item)
{
 if(Q.front == –1)
  Q.front++;
 Q.que[++Q.rear] = item;
 return Q.rear;
}
```

This condition will occur initially when queue is empty when single element will be present then both front and rear points to the same

Always increment the rear pointer and place the element in the queue

```
int delet()
{
 int item;
 item = Q.que[Q.front];
 Q.front++;
 printf("\n The deleted item is %d",item);
 return Q.front;
}
```



Linked list

- A linked list is a set of nodes where each node has two fields 'data' and 'link'. The 'data' field stores actual piece of information and 'link' field is used to point to next node. Basically 'link' field is nothing but address only.

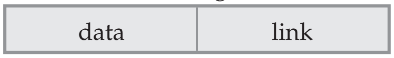


Fig. 12.4 Structure of node

- Hence link list of integers 10, 20, 30, 40 is

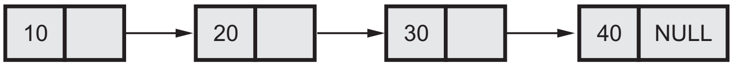


Fig. 12.5

- Note that the 'link' field of last node consists of NULL which indicates end of list.

Advantages of linked organization

1. In linked organization, insertion and deletion of elements can be efficiently done.
2. There is no wastage of memory. The memory can be allocated and deallocated as per requirement.

Disadvantages of linked organization

1. Linked organization does not support random or direct access.
2. Each data field should be supported by a link field to point to next node.

Definition of a Tree :

A tree is a finite set of one or more nodes such that -

i) Root node : It is a specially designated node.

ii) There are remaining n nodes which can be partitioned into disjoint sets T_1, T_2, T_3,... T_n where T_1, T_2, T_3,... T_n are called subtrees of the root.

- The concept of tree can be represented by following figure -

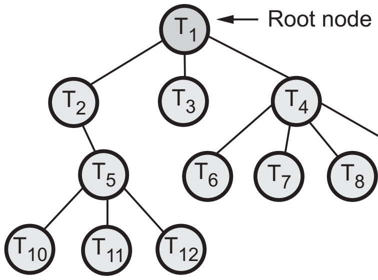


Fig. 12.6 Tree

Definition of a binary tree : *A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left sub-tree and right sub-tree.*

The binary tree can be as shown below.

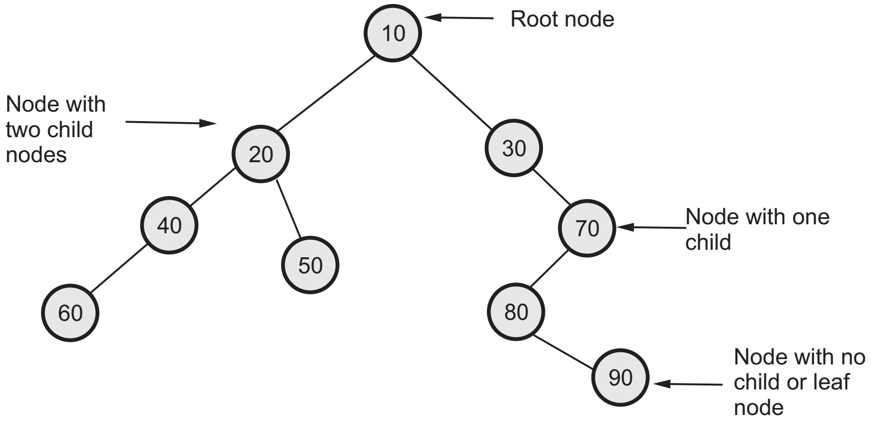


Fig. 12.7 Binary tree

❑❑❑

Notes