Computer Engineering Group

Programming in C

T. B. Kute

tbkute@gmail.com

Programming in C

(Sub Code: 9017)
Computer Engineering Group, Semester: II

First Edition
(March 2009)

T. B. Kute
(B. E. Computer)
Lecturer in Information Technology,
K. K. Wagh Polytechnic,
Nashik, India, 422003.

Windows of Knowledge are Wide and Open

Dedicated to All the Learners…….

who want to start their basics of

Programming from C

Syllabus Structure Given by Maharashtra State Board of Technical Education, Mumbai as per revised syllabus introduced from academic year 2006-2007.

Course Name: Computer Engineering Group **Course Code**: CO/CM/IF

Semester : Second

Subject Title : Programming in ‘C’ **Subject Code**: 9017

Teaching and Examination Scheme:

Teaching Scheme			**Examination Scheme**						
TH	TU	PR	Paper Hrs	TH	TEST	PR	OR	TW	TOTAL
04	--	02	03	80	20	50#	--	25@	175

Rationale:

‘C’ is the most widely used computer language, which is being taught as a core subject. C is general-purpose structural language that is powerful, efficient and compact, which combines features of high-level language and low-level language. It is closer to Man and Machine both. Due to this inherent flexibility and tolerance it is suitable for different development environments .Due to these powerful features C has not lost its importance and popularity in recently developed and advanced software industry C can also be used for system level programming, C is still considered as first priority programming language.

This subject covers from the basic concept of C to pointers in C. This subject will act as “programming concept developer” for students. It will also act as “Backbone” for subjects like OOPS, VB, Windows Programming, JAVA, OOMD, etc.

Objectives:

The students will be able to

- Describe the concepts of constants, variables, data types and operators.
- Develop programs using input and output operations.
- Write programs using different looping and branching statements.
- Write programs based on arrays and strings handling functions.
- Write programs using user-defined functions, structures and union.
- Write programs using C pointers.

Chap.	Contents	Hours	Marks
01	**Basics of C**	10	16
	1.1 History of C, where C stands 1.2 C character set, tokens, constants, variables, keywords 1.3 C operators (arithmetic, Logical, assignment, relational, increment and decrement, conditional, bit wise, special, operator precedence), C expressions data types 1.4 Formatted input, formatted output.		
02	**Decision Making and Branching**	12	24
	2.1 Decision making and branching if statement (if, if-else, else-if ladder, nested if-else) Switch case statement, break statement. 2.2 Decision making and looping while, do, do-while statements for loop, continue statement		
03	**Arrays and Strings**	14	12
	3.1 Arrays Declaration and initialization of one dimensional, two dimensional and character arrays, accessing array elements 3.2 Declaration and initialization of string variables, string handling functions from standard library (strlen(), strcpy(), strcat(), strcmp()).		
04	**Functions and Structures**	14	12
	4.1 Functions Need of functions, scope and lifetime of variables, defining functions, function call (call by value, call by reference), return values, storage classes. category of function(No argument No return value, No argument with return value, argument with return value), recursion 4.2 Structures Defining structure, declaring and accessing structure members, initialization of structure, arrays of structure.		
05	**Pointers**	14	16
	5.1 Understanding pointers, declaring and accessing pointers, Pointers arithmetic, pointers and arrays		
	Total	**64**	**80**

Chapter 01
Basics of ‘C’

Lectures allotted: 10

Marks Given: 16

Contents:

1.1 History of C, where C stands

1.2 C character set, tokens, constants, variables, keywords

1.3 C operators (arithmetic, Logical, assignment, relational, increment and decrement, conditional, bit wise, special, operator precedence), C expressions data types Formatted input, formatted output.

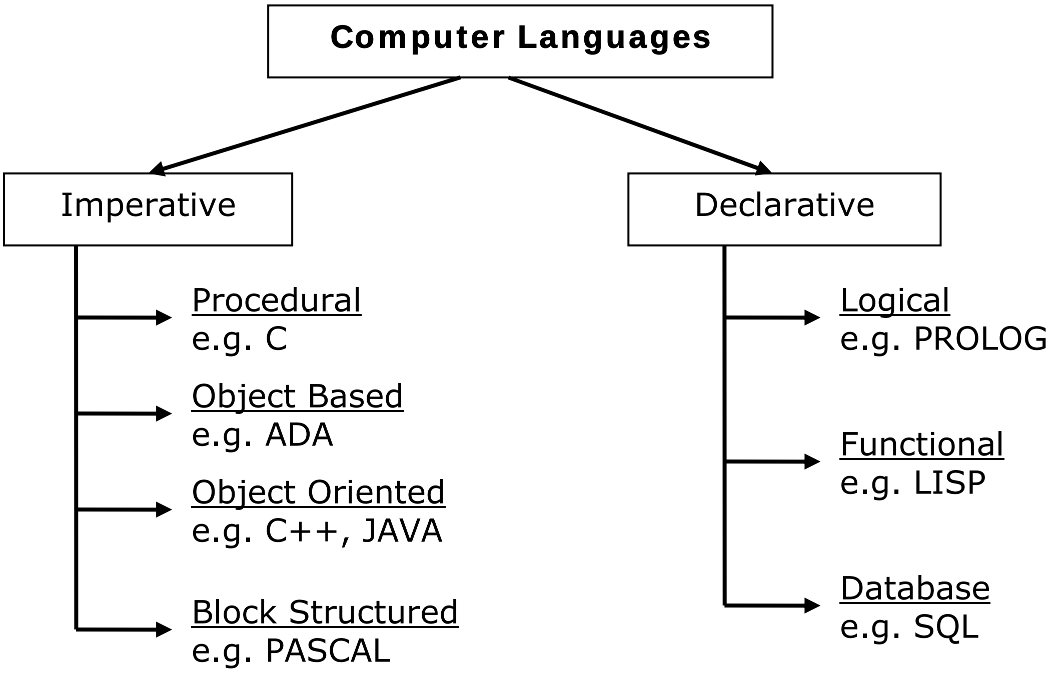
Compiler

It is system software, collection of intelligence program which converts higher level language program into machine code (Binary code). Compiler checks for syntactical errors in the program.

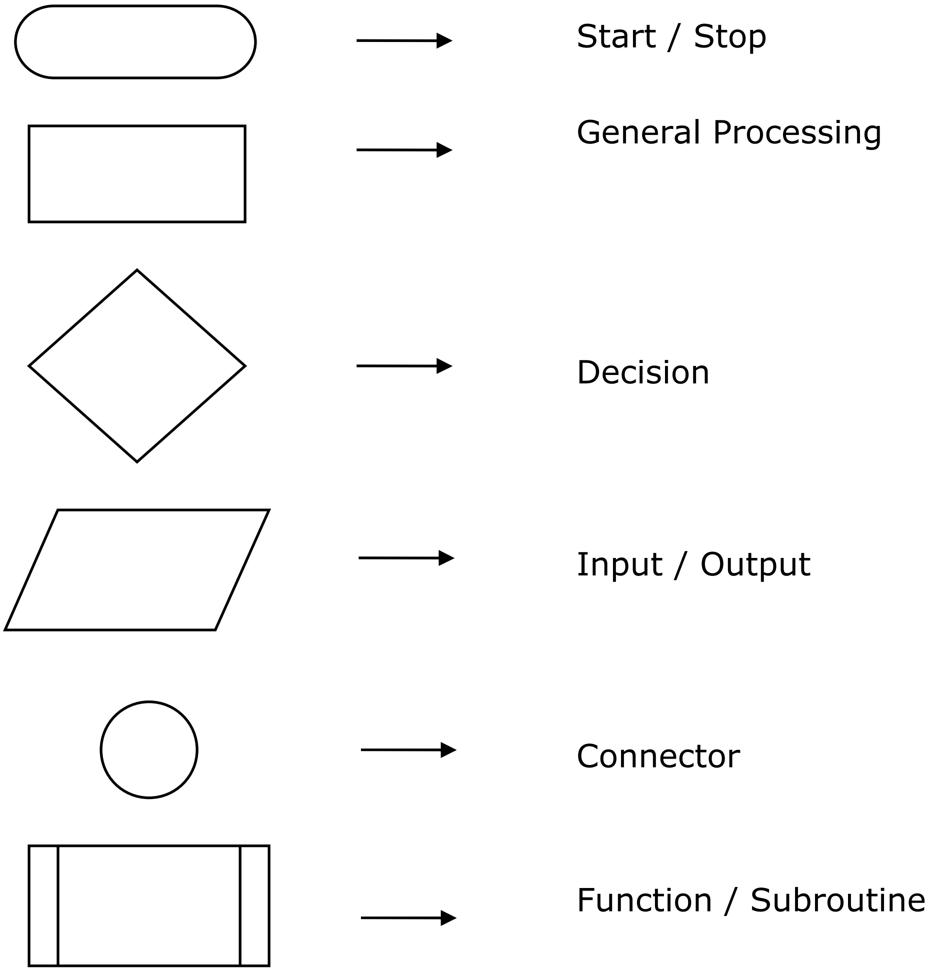
Compiler of 'C' is designed by **Dennis Ritchie** at AT&T Bell laboratory in 1974, which generally used for system programming. The name of 'C' language was derived from 'BCPL' (Basic Combine Programming Language).

Programming Languages

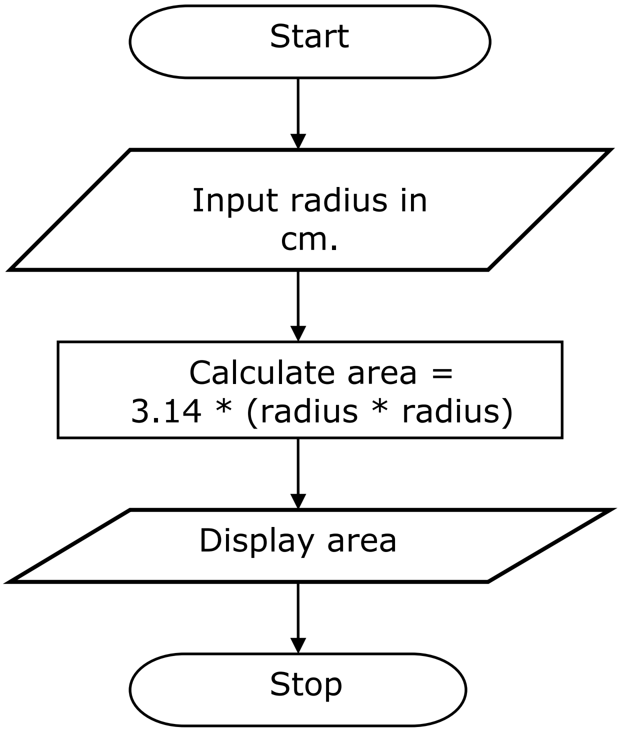
Language	**Year**	**Originator**	**Purpose**
FORTRAN	1954-57	John Backus, IBM	Numerical processing
ALGOL 60	1958-60	Committee	Numerical processing
COBOL	1959-60	Committee	Business Data Processing
APL	1956-60	K. Iverson	Array Processing
LISP	1956-62	John McCarthy	Symbolic Computation
SNOBOL	1962-66	R. Griswold	String processing
PL/I	1963-64	IBM	General purpose
SIMULA 67	1967	O. J. Dahl	General purpose
ALGOL 68	1963-68	Committee	General purpose
BLISS	1971	Wulf et. al.	System Programming
PASCAL	1971	Niklaus Wirth	General and Education
C	**1974**	**Dennis Ritchie**	**System Programming**
MESA	1974	Xerox PARC	System Programming
CONCURRENT	1975	P. Brinch	Concurrent Programming
CLU	1974-77	B. Liskov	Abstraction Methodology
EUCLID	1977	Committee	Verifiable System Programs
GYPSY	1977	Good et. al.	Verifiable System Programs
PLZ	1977	Zilog inc.	System Programming
MODULA	1977	Niklaus Wirth	System Programming
ADA	1979	J. Ichbiah	General and Embedded
C++	1983	Bjarne Stroustrup	Object Oriented Prog.
JAVA	1991	James Gosling	OOP + Embedded + N/W



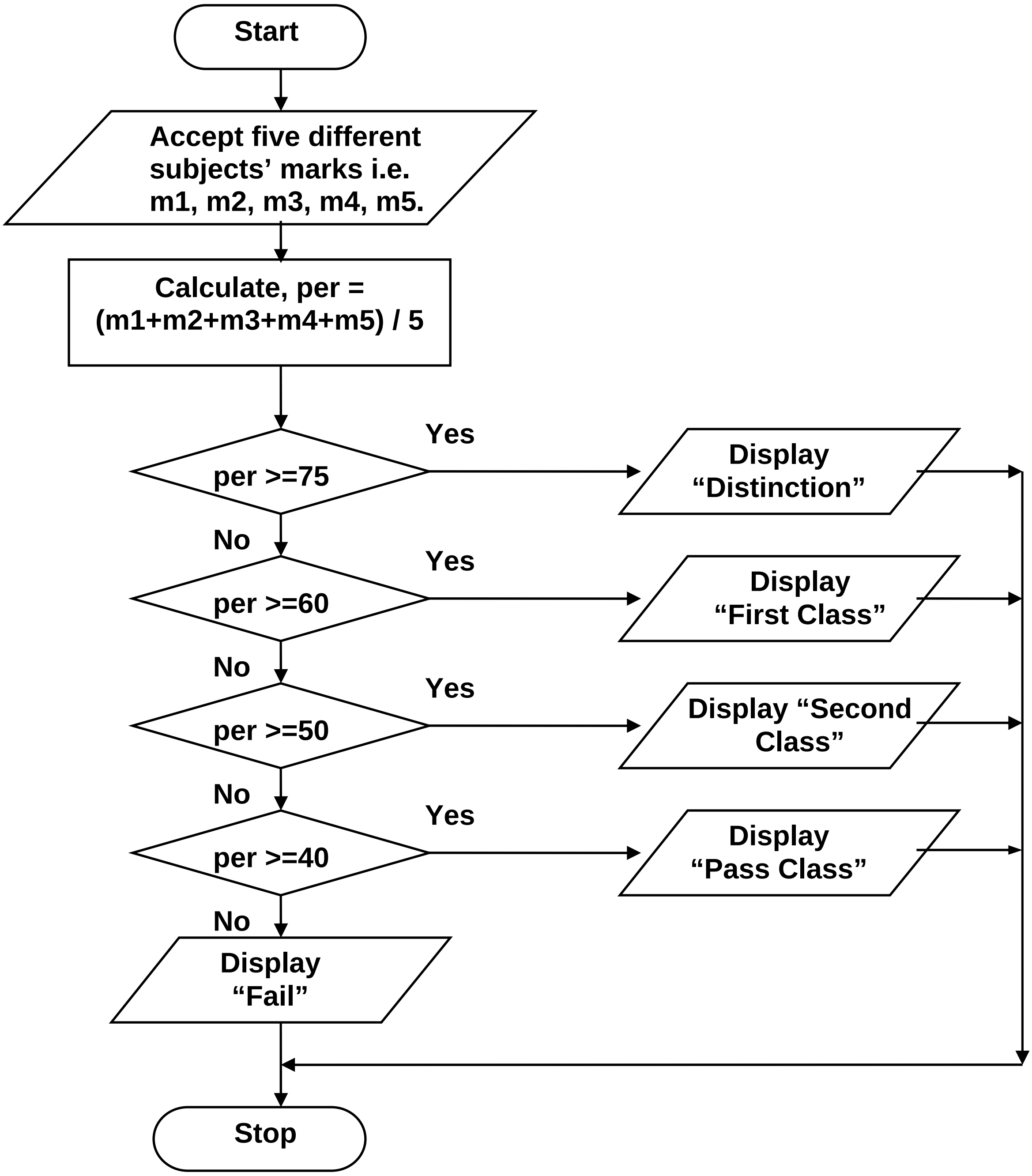
FLOW CHART SYMBOLS



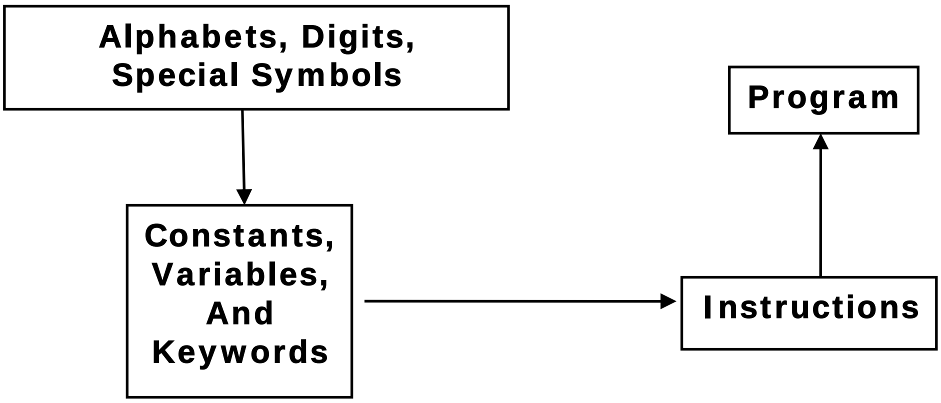
Example1: Calculating the area of the circle.



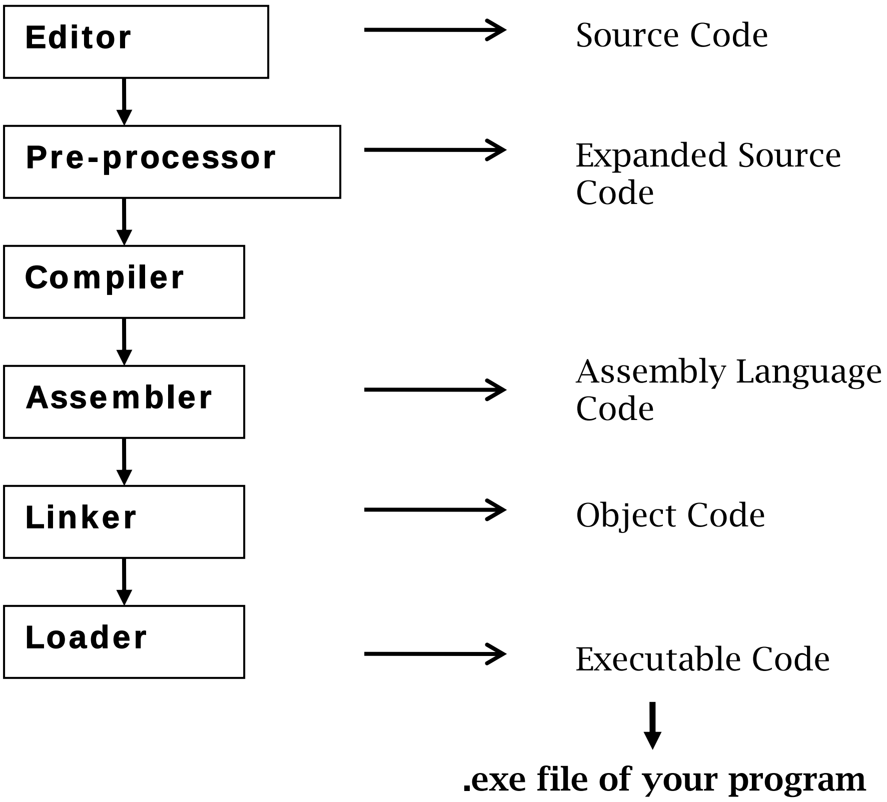
Example1: Calculating the grade obtained by students as per the marks.



Forming a C program [Ref. 2a]


Alphabets, Digits, Special Symbols
Constants, Variables, And Keywords
Instructions
Program

Compiler process in 'C'



1. **Editor**
   It is a system program in which we have written the programming code.
2. **Pre-Processor**
   The processes, which are done before the actual compilation of the program are done by the pre-processor.
3. **Compiler**
   It checks for the syntactical error in the program, if there are no errors creates the machine code of the same program
4. **Assembler**

It converts the assembly language code (Micro Processor Language) into the machine understandable code.

5. **Linker**
   It links various files of the compiler to from the object code (.obj file)
6. **Loader**
   It loads the program for memory to create the executable file.

Features of the 'C' Language

- 'C' is processor oriented structured programming language.
- It is Higher Level system programming language.
- It supports all the features of basic programming language.
- 'C' is 80% platform independent & 20% dependent.

Programming Features [Ref. 2b]

- **Character set**

  A-Z, a-z,-(under score),0-9 digits, and special symbols
  Remember 'C' is case sensitive language.
  e.g. a ≠ A

- **Constants.** [Asked in S'08]
  Primary Constants
  a. Integer Constants
  b. Real Constants
  c. Character Constants
  Secondary Constants
  a. Array
  b. Pointer
  c. Structures and Unions
  d. Enum

Primary Constants

Integer Constant:

1. Integer constant must have at least one digit.
2. They must not have a decimal point.
3. They can be positive or negative.
4. If there is no any sign to integer it is assume to be positive.
5. No comma or blank spaces are allowed within the integer constant.
   e.g. 426, +756,-588 etc.

Real Constant:

1. These are generally called as floating point constant. They can be written two forms. That is, fractional form & exponential form.
2. Real constant can be positive or negative.

3. They must have decimal point.
4. Default sign is positive.
5. No comma or blank spaces are allowed in the real number.
   e.g. 22.45, -85.23, -11.20, +3.211e-4, 5.6e4 etc.

Character Constants:

1. The character constant is a single alphabet or a single digit or a single special symbol enclosed within the single quotation marks.
2. Maximum length of a character is one character.
   e.g. 'f', 'h', '=', '8' etc.

- **Variables**

[Asked in W'07, S'08, W'08]

It is an entity that may vary during the program is executing. Variable is the name given to the location of the memory. This location can contain integer, real or character constants.

Rules for creating variable's name:

1. Variable name is the combination of alphabets, digit & underscore.
2. First character in the variable name must be an alphabet.
3. A keyword can not be a variable.
4. No commas, blanks and special symbols are allowed in the variable name.

- **Keywords**

[Asked in S'08]

Keyword is the word who's meaning as already being given to the 'C' compiler. The keywords can not be given as variable name in 'C'. These are also called as reserve words. The C supports 32 different keywords listed below:

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
while	volatile	near	for	asm	

- **Operators** [Ref. 1a]

Operators are the special symbol use to perform special operation (for example, arithmetic & logical) on operands. Operands are the variables, constants or expressions on which operators perform operation.

As per the number of operands required by the operator, it is classified in three categories:

1. Unary operators
2. Binary operators
3. Ternary operators

These operators require one, two and three operands respectively for performing their operation.

Numerical or Arithmetic Operators:

Operator	Type	Meaning	Example
+	Binary	Addition	9 + 45
–		Subtraction	89 – 12
*		Multiplication	10 * 3
/		Division	18 / 6
%		Modulo Division	14 % 3
+	Unary	Unary Plus	+51
–		Unary Minus	–92
=	Binary	Assignment	a = 45 + 10/2
++	Unary	Increment [Asked in S'W'07]	Post increment: a++ Pre increment: ++a
––		Decrement [Asked in S'W'07]	Post decrement: a–– Pre decrement: ––a

+=	Binary	Add and assign	a += 10 implies that
			a = a + 10

–=		Subtract and assign	a –= 10 implies that a = a – 10
*=		Multiply and assign	a *= 10 implies that a = a * 10
/=		Divide and assign	a /= 10 implies that a = a / 10
%=		Mod and assign	a %= 10 implies that a = a % 10

Last five operators are also called as assignment operators.

Relational Operators

[Asked in S'W'08]

These operators are used to check the relation between the operands.

Operator	Type	Meaning
<	Binary	Less than
>		Greater than
<=		Less than or equal to
>=		Greater than or equal to
==		Equal to
!=		Not equal to

Logical Operators

[Asked in S'07'08, W'08]

These operators are used to combine two conditions.

Operator	Type	Meaning	Use
&&	Binary	Logical AND	Combines two condition to check whether both are true or not
||		Logical OR	Combines two condition to check whether any one of these is true or not
!	Unary	Logical NOT	Inverses the condition

Conditional Operator

[Asked in W'08]

The only ternary operator (operator having three operands) is defined in 'C' called as conditional operator. The character pair ? : is termed as conditional operator. This is used to construct the conditional expression of the following form:

```
condition   ?   expression2   :   expression3
```

When the condition written is true then the expression1 is evaluated else expression2 is executed. Means, any one of the expression1 and 2 is executed in any case. It can also be called as if-then-else operator.

```
x = 45;
y = 22;
(y>25) ? x = y : x = 50;
```

After completion of third statement execution, value of x will become 50. Because, the condition y>25 is false so first expression will not be executed. Only second expression is evaluated. That is, 50 will be assigned to variable x.

Bitwise operators

[Asked in S'W'07]

Operator	Type	Meaning
&	Binary	Bitwise AND
|		Bitwise OR
^		Bitwise EX-OR
~	Unary	Bitwise NOT
<<	Binary	Left shift
>>		Right shift

Special Operators

Operator	Type	Meaning
.	Binary	Member selection operator
#	Unary	Preprocessor operator
->	Binary	Pointer member selection operator
\	Unary	Escape sequence operator

:	Unary	Used for case value declaration as well as goto label declaration
sizeof	Binary	Finding size of data or variable
(type)	Unary	Type casting

- **Separators**

These are the symbols which are used to indicate where groups of codes are divided and arranged. They are basically used to define the shape and function of the program code. The most commonly used separator is semicolon. All these separators are listed below:

Separator	Name	Purpose
( )	Parenthesis	Used to contain lists of parameters in function definition and call. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Curly Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code for classes, functions, and local scopes.
[ ]	Square brackets	Used to declare array types. Also used when dereferencing array values.
< >	Angle Brackets	Used to include header files in the program.
;	Semicolon	Terminating the statement/line
,	Comma	Separates consecutive variable declarations. Also used to chain statements together inside a ‘for’ loop statement.
“ ”	Double quotes	Used to declare any string or group of more than one characters
‘ ’	Single quotes	Used to declare a single character constant

- **Comments**

Comments are the statements written inside /*…….*/ will not be executed by the compiler. Compiler will not check for any syntactical or logical error for the statement written inside the comments. There are two types of comments used in C.

1. /*……*/ multi-line comments
2. // single line comments

Arithmetic Expressions [Ref. 2c, 1b]

An arithmetic expression is combination of variables, constants and operators as per the syntax of the ‘C’ programming language. ‘C’ can handle any complex arithmetic expression. But, we have to take care of converting it into the format and appropriate syntax of the 'C' language.

Table gives some examples of arithmetic expressions.

Arithmetic Expression	'C' expression
$(x+y)^2$	(x + y) * (x + y)
$x^2 + y^2 + 5$	x * x + y * y + 5
$(xy)(m-n)$	x * y * (m – n)
$\frac{ab}{c}$	a * b / c
$\frac{x}{Y} + c$	(x / y) + c

These expressions are evaluated using an assignment statement of the form:

```
variable = expression;
```

Here, 'variable' is a valid 'C' variable name. When the statement is encountered, the expression is evaluated first. The result then replaces previous value of variable on the left-hand side (referred as Lvalue). All the variables used in the expressions must be assigned values before evaluation is attempted.

For example:

```
a = (x + y) * (x + y);
b = (x / y) + c;
x = a * b / c;
```

Precedence and Associativity of operators [Ref. 2d, 1c]

When the expression involves more than one operation, then which operation is to be performed first is decided by the precedence of that operator. Highest precedence operation is solved first. Each operator of 'C' has precedence associated with it. The operators with the same precedence are evaluated according to there associativity. That is, the expression is evaluated either from left to right or right to left. The precedence and associativity of all these operators is given in the following table.

No.	Operator	Description	Asso.
1	() [] . -> ++ --	Parentheses (grouping) Brackets (array subscript) Member selection via object name, Member selection via pointer, Postfix increment / decrement [Asked in S'07]	L to R
2	++ -- + - ! ~	Prefix increment/decrement [Asked in S'07] Unary plus/minus	R to L

	(*type*) * & sizeof	Logical negation/bitwise complement Cast (change *type*) Dereference Address Determine size in bytes	
3	* / %	Multiplication/division/modulus	L to R
4	+ -	Addition/subtraction	L to R
5	<< >>	Bitwise shift left, Bitwise shift right	L to R
6	< <= > >=	less than/less than or equal to greater than/greater than or equal to	L to R
7	== !=	Relational is equal to/is not equal to	L to R
8	&	Bitwise AND	L to R
9	^	Bitwise exclusive OR	L to R
10	|	Bitwise inclusive OR	L to R
11	&&	Logical AND	L to R
12	||	Logical OR	L to R
13	?:	Ternary conditional	R to L
14	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition / subtraction assignment Multiplication / division assignment Modulus / bitwise AND assignment Bitwise exclusive / inclusive OR assignment Bitwise shift left/ right assignment	R to L
15	,	Comma (separate expressions)	L to R

Hierarchy of operations [Ref. 2e]

```
i = 5 * 6 - 8 + 9 - (43 + 2) /5 - 5;
i = 30 - 8 + 9 - 45/5 - 5;
i = 30 - 8 + 9 - 9 - 5;
i = 22 + 0 - 5;
i = 17;
```

Data types

[Asked in S'W'07, S'08]

```
     int a;                 /* General declaration */
     int a = 10;            /* Valued Declaration */
     float b = 12.6;
     float b;
     char c;
     char c = 'y';
```

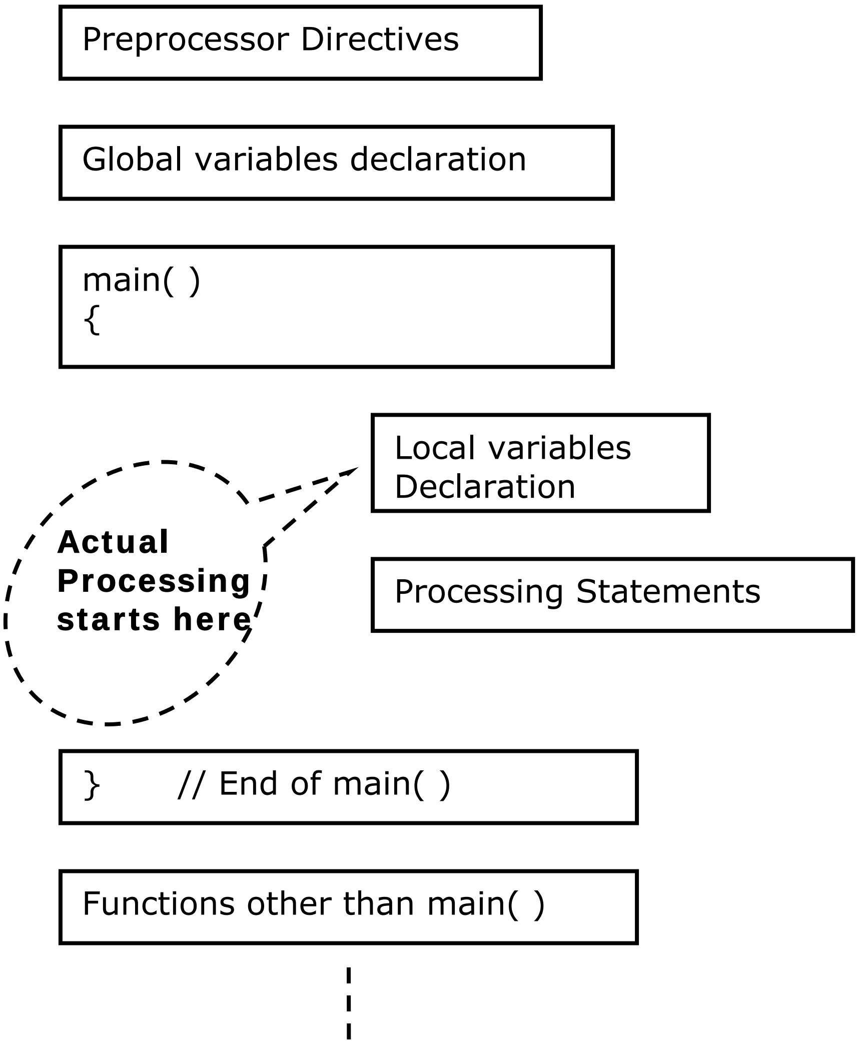
Maximum values of Data Types [Ref. 1d]

- **int** requires 2 bytes to store in memory.
  -32768 to 32767
- **float** requires 4 bytes to store.

-3.4e38 to 3.4e38

- **char** requires 1 byte to store.
  A….Z and a….z also special symbols
- **long int** requires 4 bytes to store.
  -2147483648 to 2147483647.
- **double** requires 8 bytes to store.
  -1.7e308 to 1.7e308
- **long double** requires 10 bytes to store.
  -1.7e4932 to 1.7e4932

Structure of a C program [Ref. 1e]


Preprocessor Directives
Global variables declaration
main( )
{
Actual Processing starts here
Local variables Declaration
Processing Statements
}   // End of main( )
Functions other than main( )

Formatting input and output statements [Ref. 1f, 1g]

1. printf

[Asked in S'08]

This formatted output function statement defined in stdio.h file used to print the string or the values of variables on the output screen.

Syntax:

```
     printf("<format string>", <variables-list>);
```

Here, the format string will contain the string and the format specifiers or format codes to display the values of variables in the sequence of format specifiers.
e.g.

```
    printf(“Welcome to C”);
```

it will print following string on the output screen.

```
Welcome to C
```

In order to print the values of the variables printf can also be used as:

```
    int y = 14, s = 10;
    printf(“Values are : %d and %d”,s, y);
```

this will print the following statement on the screen:

```
Values are: 10 and 14
```

Here the %d is called as integer format specifier and used to print the integer value of variable on the screen. Such 16 different format specifiers / format codes are used in printf as well as scanf. These are listed below:

Format Specifiers / format codes [Ref. 1h]

[Asked in S’W’08]

%c	Character format specifier.
%d	Decimal integer format specifier.
%e	Scientific notation for floating point format specifier.
%E	Scientific notation for floating point format specifier.
%f	Floating-point format specifier.
%g	Uses %f or %e, whichever result is shorter.
%G	Uses %f or %E, whichever result is shorter.
%i	Integer format specifier (same as %d).
%n	Records the number of characters written so far.
%o	Unsigned octal format specifier.
%p	Displays the corresponding argument that is a pointer.
%s	String format specifier.
%u	Unsigned integer format specifier.
%x	Unsigned hexadecimal format specifier.
%X	Unsigned hexadecimal format specifier.
%%	Outputs a percent sign (%).

Escape Sequences [Ref. 1i]

Character	Meaning
\n	New line
\t	Tab
\b	Backspace
\v	Vertical tab
\a	Audible Alert (Bell)
\r	Carriage return
\f	Farm Feed
\\	Backslash
\‘	Single Quote
\“	Double Quote

2. scanf

[Asked in S’08]

This formatted input function statement defined in stdio.h file and used to accept the values of variables from the keyboard.

Syntax:

```
    scanf(“<format string>”, <variables-list>);
```

Here, the format string will contain the format specifiers or format codes to accept the values of variables in the sequence of format specifiers.

For example:

```
    int var1;
    float var2;
    scanf(“%d%f”, &var1, &var2);
```

This statement will read an integer number and a float number from keyboard and store these values in var1 and var2 respectively. The scanf allows all the above format specifiers to use for inputting the values.

- **Type casting or Type conversion** [Ref. 1j]

The C programming language has ability to convert one type of the data into another type tempoparily. This is called as type casting or type conversion. When a C expression contains different types of the data, the result of that expression is always be in highest data types in the expression. For example:

```
int num1, num2;
float x;
long y;
```

here, when we write the expression,

```
num1 * num2 + x/y;
```

will always result in the long type of the data. And the expression,

```
num2 + x * num1;
```

will result in float type of the data. This is because, it is the property of the expression. It is referred as automatic type casting. The type cast operator - ( ) can be used to convert one data type into another temporarily. This is referred as manual type casting. Such as:

```
int x = 10, y = 4;
float result;
result = x / y;
printf("Division: %f", result);
```

Here, the printf statement will output:

```
Division: 2.000000
```

But, it is not the accurate answer. So type casting is required. Such as,

```
result = (float) x / y;
```

This statement will convert value of x temporarily into float type. So result of expression will automatically be float. Now the printf statement will output:

```
Division: 2.500000
```

This concept is referred as type casting, which is found to be one of the useful concepts in 'C'.

Sample Programs:

1) Basic salary of employee is input through the keyboard. DA is 40% of basic salary & house rent allowance (HRA) is 20% of the basic salary. Calculate his gross salary?

```
Ans :   #include<stdio.h>
        #include<conio.h>
        void main( )
        {
          float Basic, DA, HRA, GS;
          clrscr( );
          printf("Enter the basic salary");
          scanf("%f", &basic);
          DA = (40/100) * basic;
          HRA = (20/100)* basic;
              GS = DA + HRA + Basic;
          printf (“Gross salary = %f ”,GS);
          getch ( );
       }
```

2) Temperature of a city is Fahrenheit degrees is input to keyboard write a program to convert this temperature into centigrade.
    [TC = (TF-32)*5/9]

```
Ans :   #include<stdio.h>
        #include<conio.h>
        void main( )
        {
          float centi;
          int faren;
          printf(“Enter Fahrenheit temp”);
          scanf(“%d”, &faren);
          centi = (( float) faren -32) * 5 / 9;
          printf(“ cent temp = %f”, cent);
          getch( );
        }
```

3) The length and breadth of the rectangle & radius of the circle are input through key board. Write a program to calculate the area, perimeter & the circumference of the circle.

```
Ans :    #include<stdio.h>
         #include<conio.h>
         void main( )
         {
            float  l, b, r, area, perim, circumf;
            clrscr( );
            printf(“Enter length & breadth”, l, b,);
            scanf(“%f%f”, &l,&b);
            area = l * b;
            perim = 2 * (l + b);
            printf(“\n area =%f ”, area);
            printf(“\n perim = %f ”, perim);
            printf(“\n enter radius ”,r);
            scanf(“%f ”, &r);
            area = 3.14 * r * r;
            circumf = 2 * 3.14 * r;
            printf(“\n area = %f ”,area);
            printf(“\n circumf = %f ”, circumf);
            getch( );
        }
```

4) Two numbers are input through keyboard into two locations C & D. Write a program to interchange the content of C & D.

```
Ans :   #include<stdio.h>
        #include<conio.h>
        void main( )
        {
            int a, b, c;
            clrscr( );
            printf(“a = %d, b = %d ”, a, b);
            c = a;
            a = b;
            b = c;
            printf(“\n a = %d, b = %d”, a, b);
            getch( );
        }
```

Review Questions

Summer 2007 **Marks**

1. State different data types supported by 'C' language (W-2007). 2
2. Write a 'C' statement for each of the following: 2
   i. Relational operator.
   ii. Assignment operator.
3. State the use of increment and decrement operator and give its precedence and Associativity. 4
4. Define the following: 4
   i. Expressions
   ii. Data types
   iii. Bitwise operators
   iv. Logical operators

Winter 2007

1. Explain the use of bitwise operator. 2
2. State the use of increment and decrement operators. Also give difference between i++ and ++i with example. 4
3. What is variable? How to declare it? How variable differs from constant? 4

Summer 2008

a) What is variable declaration and variable initialization? 2

b) What are the uses of printf( ) and scanf( )? 2

c) List different relational and logical operators. What will be the output of following program? 4

```
#include<stdio.h>
void main( )
{
     float a = 5, b = 2;
     int c;
     c = a % b;
     printf("%d",c);
}
```

d) State use of %c %d and %f. 4

Write output of following:

```
#include<stdio.h>
main( )
{
     float y = 123.456;
     printf("%f %.3f %.1f \n\n", y, y, y);
}
```

e) Define the following: 4

1) Keyword
2) Variable.
3) Data types
4) Constants.

Winter 2008

a) State four relational operators with its meaning. 2

b) State four printf format codes. 2

c) State the arithmetic and logical operators with its meaning. 4

d) Write a program to take marks of five subjects and display the total and average. 4

e) State four rules of choosing variable's name. 4

f) What is the output of the following code? 4

```
void main( )
{
    int i = 1, j = -1, k = 0, w, x, y, z;
    w = i || j ||k ;
    x = i && j && k ;
    y = i || j && k ;
    z = i && j && k ;
    printf(« w=%d x=%d y=%d z=%d »,w,x,y,z) ;
}
```

Other:

1. Differentiate between unary, binary and ternary operators. 4
2. What is precedence and associativity of operators? 2
3. Explain the use of manual and automatic type casting with suitable. Example. 4
4. What is comment? Explain its use. 2

Programming Exercises

1. Write a program to input the temperature in Celsius and display its equivalent Fahrenheit temperature value.
2. The distance between two cities (in km.) is input through the keyboard. Write a program to convert and print this distance in meters, feet, inches and centimeters.
3. Write a program to input the radius of the circle in centimeters and find the area of that circle.
4. Write a program to input the height and base of the right angle triangle and find the hypotenuse of it using Pythagoras' Theorem.
5. Input to the program is length and width of rectangle. Find the area and perimeter of it.
6. Two numbers are input through the keyboard into two locations C and D. Write a program to interchange the contents of C and D.

References

1. **Programming in ANSI C**
   by, E Balagurusamy, Tata McGraw Hill, 4th Edition
   a. Chapter 3, Page 52
   b. Chapter 3, Table 3.6, Page 64
   c. Chapter 3, Table 3.8, Page 73
   d. Chapter 3, Table 2.7, 2.8, Page 32
   e. Chapter 1, Fig. 1.3, 1.9, Page 6, 13
   f. Chapter 1, Topic 1.3, Page 3
   g. Chapter 2, Topic 2.10, Page 38
   h. Chapter 4, Table 1.4, Page 104
   i. Chapter 2, Table 2.5, Page 29
   j. Chapter 3, Topic 3.14, Page 68

2. **Let us C**
   By, Yashwant Kanetkar, BPB Publications, 7th Edition
   a. Chapter 1, Fig. 1.1, Page: 5
   b. Chapter 1, Page: 5
   c. Chapter 1, Fig. 1.19, Page: 35
   d. Appendix A, Page 707
   e. Chapter 1, Example 1.1, Page 33

Chapter 02

Decision Making and Branching

Lectures allotted: 12

Marks Given: 24

Contents:

Decision Making Statements

Many times in the program such conditions occur when we want to take the exact decisions in order to make an error free program. This kind of situations can be handled using decision control instructions of C. It includes if - else statements and the conditional operators. The decision control structure in C can be implemented using:

a) The if statement
b) The if - else statement
c) Nested if – else
d) The else – if ladder

The if Statement [Ref. 1a, 2a]

The general form or syntax of if statement looks like this:

```
if (condition)
{
     //execute these statements;
}
```

Here, the keyword ‘if’ tells the compiler that what follows, is a decision control instruction. The condition following the keyword **if** is always enclosed within a pair of parentheses. If the condition is true, then the statements in the parenthesis are executed. It the condition is not true then the statement is not executed instead the program skips this part. The condition in C is evaluated using C’s relational operators as well as logical operators. These operators help us to build expression, which are either true or false.
For example:

```
X == Y    X is equal to Y
X != Y    X is not equal to Y
X < Y     X is less than Y
X > Y     X is greater than Y
X <= Y    X is less than or equal to Y
X >= Y    X is greater than or equal to Y
```

Demonstration of if statement:

```
/* Example 2.1 */
#include<stdio.h>
main( )
{
     int num;
     printf(“Enter a number less than 10 :”);
     scanf(“%d”, &num);
     if(num < = 10)
```

```
                printf(“The number is less than 10”);
        }
```

Sample output of this program will be:

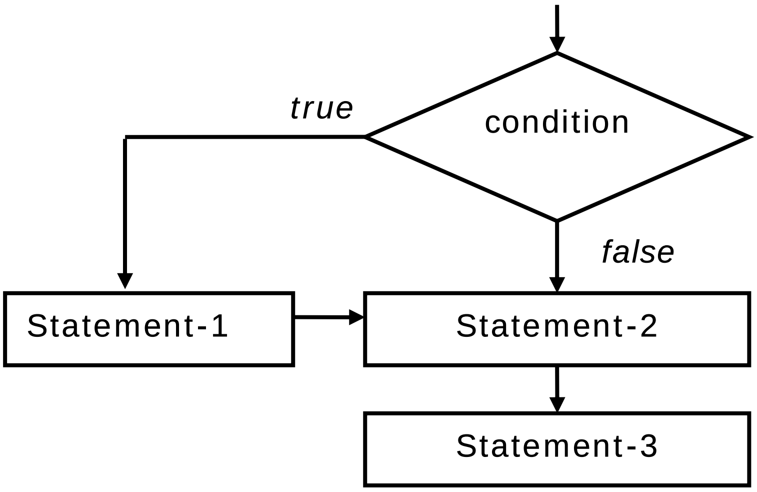
```
Enter a number less than 10 : 7
The number is less than 10
```

Executing this another time:

```
Enter a number less than 10 : 15
```

After second statement there will no output because the condition given in the ‘if’ statement is false. The ‘if’ statements are only executed when the condition is true.

Flowchart [Ref. 1b]



See the above flowchart. If the condition is true then statement-1 is executed else and if false it won’t be executed. Remember, statements2 and 3 are executed always. The ‘condition’ can be the combination of relational and logical operators also. Such as:

```
if (i > 1 || j!=5)
{
     printf("i = %d “, i );
}
```

Multiple Statements within if [Ref. 2b]

If it is desired that more than one statement is to be executed if the condition following if is satisfied, then such statements must be placed within pair of braces.
e.g The following program demonstrate that if year of service greater than 3 then a bonus of Rs. 2500 is given to employee. The program illustrates the multiple statements used within if.

```
/* Example 2.2 calculation of bonus */
#include<stdio.h>
```

```
main( )
{
     int bonus, cy, yoj, yos;
     printf(“Enter current year and year of joining : ”);
     scanf(“%d%d”,&cy,&yoj);
     yos = cy - yoj;
     if(yos > 3)
     {
          bonus = 2500;
          printf(“Bonus = Rs. %d”, bonus);
     }
}
```

Let’s see the output of this program:

```
     Enter current year and year of joining : 2006 2001
     Bonus = Rs. 2500
```

In this program, value of cy and yoj are taken from user i.e. 2006 and 2001. Value of yos will become 5. The condition within ‘if’ will check whether the value of yos is greater than 3 or not. Here, condition will become true and the statements followed by if are executed.

The if-else statement [Ref. 1c, 2c]

[Asked in S’07]

The ‘if’ statement by itself will execute a single statement or a group of statements when the condition following if is true, it does nothing when the condition is false. If the condition is false then a group of statements can be executed using ‘else’ statement. Its syntax is as given below:

```
     if(condition)
     {
          //statements1
     }
     else
     {
          //statements2
     }
```

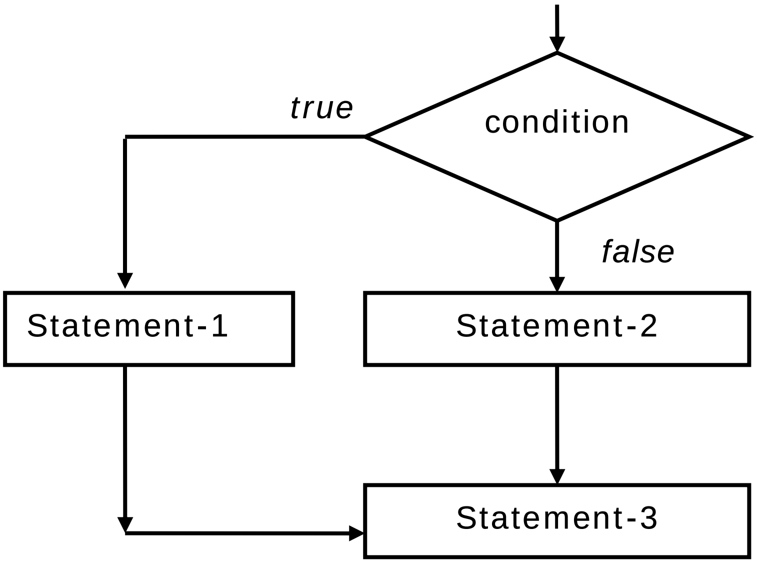
Here, if the condition is true then statements1 are executed else statements2 are executed. It is demonstrated in following program.

[Ref. 2d]

```
/* Example 2.3 Calculation of gross salary */
#include<stdio.h>
main( )
{
     float bs, gs, da, hra;
     printf(“Enter basic salary”);
     scanf(“%f”, & bs);
     if(bs <1500)
     {
          hra = bs * 10/100;
          da = bs * 90/100;
     }
     else
     {
          hra = 500;
          da = bs * 98/100;
     }
     gs = bs+hra+da;
     printf("gross salary = Rs. %f", gs);
}
```

When, we used if – else statements, either 'if' or 'else' statement block will execute depending upon the condition given in the 'if' statement. The 'else' doesn't require condition. Execution of 'else' block also depends on `if(condition)`. When it is false program control transfers to 'else' block.

Flowchart [Ref. 1d, 2e]



Note:

1. Statements inside the 'if's curly braces are called as 'if block' and statements for else's curly braces are called as 'else block'.
2. There is no necessity to have an else for if but, each else must match an if statement else there will be a compiler error. The error will be 'Misplaced else'.
3. If there is only one statement inside 'if' as well as 'else' block, we can eliminate the curly braces.

Nested if – else [Ref 1e, 2f]

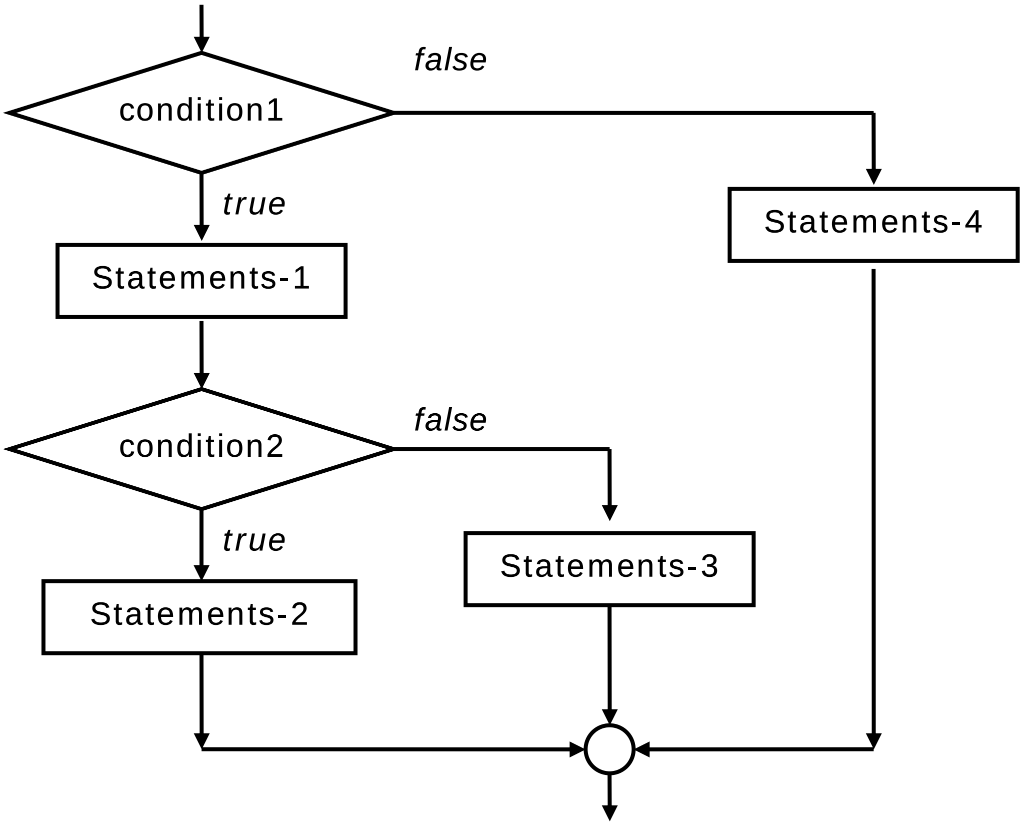
[Asked in S'07, W'07, S'08]

If we write an entire if - else construct within the body of the 'if' statement or the body of an 'else' statement. This is called 'nesting' of if-else.

Syntax:

```
if(condition)
{
     //statements1
     if (condition)
     {
          //statements2
     }
     else
     {
          //statements3
     }
}
else
     //statements4
```

Here, the inner condition executed only when the outer condition of 'if' is true. This hierarchy of nesting of 'if' can be extended in deep with any number of 'if-else' statements.



Observe the following example.

```
/* Example 2.4 */
#include<stdio.h>
main( )
{
     int y;
     printf(“Enter year : “);
     scanf(“%d”, &y);
     if(y > 2000)
     {
          printf(“\n Number is greater than 2000”);
          if(y % 4 == 0)
          {
               printf(“\n This is leap year”);
          }
          else
          printf(“\n This is not leap year”);
     }
     else
     {
          printf(“\n Number is less than 2000);
          if(y % 4 ==0)
          {
               printf(“\n This is leap year”);
          }
          else
          printf(“\n This is not leap year”);
     }
```

Output:

```
     Enter year : 1988
     Number is less than 2000
     This is leap year
```

When the user enters 1988, first if condition checks whether it is greater than 2000 or not? Here this condition will become false and control will transfer to else block. Again if(y % 4==0) inside else block will check whether it is totally divisible by 4 or not? If it is yes, ‘if block’ will be executed else ‘else block’ will be executed.

Several forms of if – else [Ref. 2g]

[Asked in S’08]

```
  a)  if (condition)
           //perform this;

  b)  if (condition)
      {
        //perform this;
        //as well as this;
    }

c)  if (condition)
        //perform this;
    else

        //perform this;

d)  if (condition)
    {
        //perform this;
        //as well as this;
    }
    else
    {
        //perform this;
        //as well as this;
    }
e)  if (condition)
    {
        //if (condition)
        //perform this;
    }
    else
    {
        //perform this;
        //as well as this;
    }
f)  if (condition)
    {
        if (condition)
             //perform this;
        else
             //perform this;
    }
    else
    {
        if (condition)
             //perform this;
        else
             //perform this;
    }
```

Using Logical operators [Ref. 2h]

As we have already seen the logical operators. The basic use of these is related to if – else statement. These conditional operators are:

a) && (AND operator)

b) || (OR operator)

c) ! (NOT operator)

When, we want to specify more than one condition in the single if statement we can use logical operators among these conditions. Though, there are two & (and) in the operator it is pronounced as AND and || as OR. && Operator is used when there is necessity that both the conditions to be true. || Operator is used when there is necessity that any one conditions to be true. e.g.

```
if (a > 15 && b < 10)
{
    printf("Hello");
}
```

Here, if value of 'a' is greater than 15 as well as value of 'b' is less than 10 then printf("Hello"); statement will be executed. Else it will perform the statements after the if block. Similarly, see the example of OR operator.

```
if (a > 15 || b < 10)
{
    printf("Hello");
}
```

Here, if value of 'a' is greater than 15 or value of 'b' is less than 10 then printf("Hello"); statement will be executed. Else it will perform the statements after the 'if' block.

Now, let's focus on the NOT operator (!). This operator reverses the value of expression it operates on; it makes true expression false and false expression true. e.g.

```
! (a > 10)
```

This means that whether 'a' is not greater than 10. In other words, it can be written as (a <= 10).

The else-if ladder [Ref. 1f, 2i]

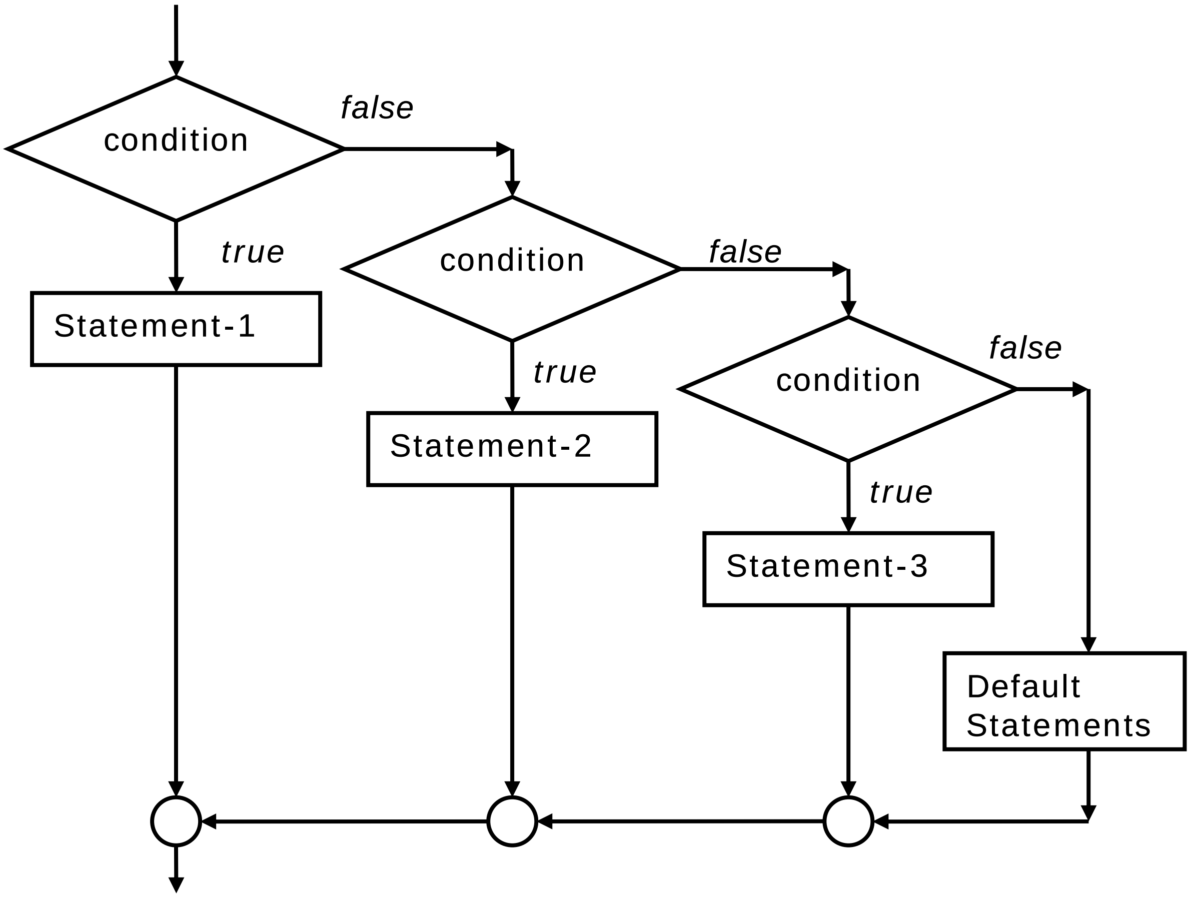
[Asked in W'08]

There is another way of putting multiple 'if's together when multi-path decisions are involved. A multipath decisions is a chain of 'if's in which the statement is associated with each else in an 'if'. It takes the following general form:

```
if(condition1)
     //statements1;
else if(condition2)
          //statements2;
     else if(condition3)
               //statements3;
          else if(condition n)
                    //statements n;
               else
```

```
                    //statements-default;
//statements-x;
```

This construct is known as **else-if ladder**. The conditions are evaluated from top to the bottom of ladder. As soon as true condition is found, statements associated with it are executed and then program control is transferred to the 'statements-x', skipping the rest of the ladder. When all the conditions become false, then final else containing default statements will be executed. This is shown in the following flowchart [Ref 1g].



See the practical example on this.

```
/* Example 2.5 */
#include<stdio.h>
main( )
{
     int a, b, c, d, e, per;
     printf("Enter marks of four subjects : ");
     scanf("%d %d %d %d", &a, &b, &c, &d);
     if(a < 40 || b < 40 || c < 40 || d < 40)
          printf("\n You have failed in one subject");
     per = (a + b + c + d) / 4;
     if(per > = 60)
          printf("\n First class");
     else
     {
      if (per <=59 && per >=55)
          printf("\n Higher second class");
      else
          if(per <=54 && per >=40)
               printf("Second class);
          else
               printf("\n Fail");
     }
}
```

In this example we have used else – if ladder as well as logical operators.

```
if(a < 40 || b < 40 || c < 40 || d < 40)
```

This statement will check whether marks of any one subjects is less than 40 or not? If it is, it will execute the statement followed by 'if' i.e.

```
printf("\n You have failed in one subject");
```

Now, see the following statement,

```
if (per <=59 && per >=55)
```

Here, if the value of 'per' is less than or equal to 59 as well as greater than or equal to 55 then the statement followed by this will be executed i.e.

```
printf("\n Higher second class");
```

Else, the control will move to the else block.

The 'switch' Statement [Ref. 1h, 2j]

[Asked in W'07, S'08, W'08]

This is another form of the multi-way decision statement. It is well structured, but can only be used in certain cases. The switch statement tests value of a given variable (or expression) against the list of case values and when the match is found, a block of statements associated with that statements are executed. The general form of switch statement is as follows:

```
switch(variable)
{
     case value-1:
                    //statements1;
                    break;
     case value-2:
                    //statements2;
                    break;
     case value-3:
                    //statements3;
                    break;
     - - - - -
     - - - - -
     default:
                    //default-statements;
                    break;
}
statements-x;
```

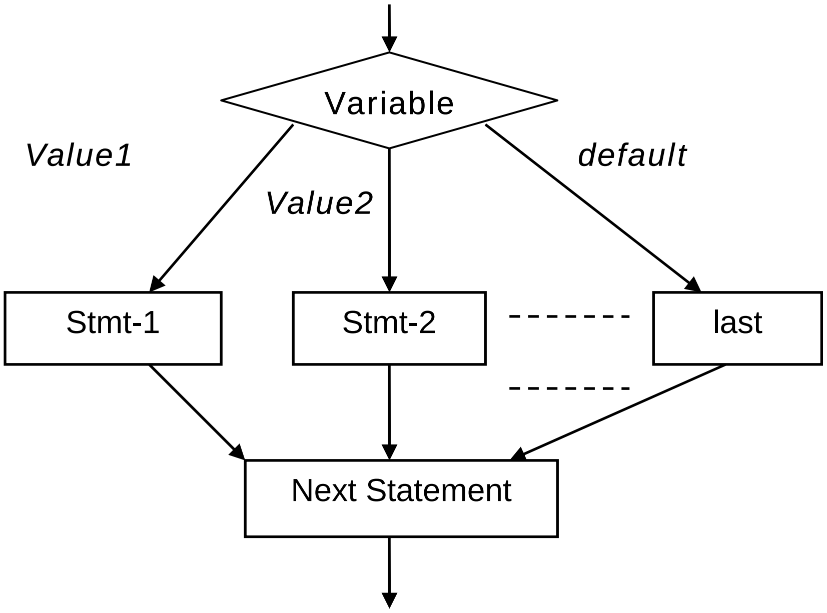
In the above given syntax, value-1, value-2, value-3…. are the set the constants. When the value of the variable given in the switch brackets is matched with any one of these values, the particular set of statements are executed and then program control transfers out of the switch block. For example, if value of variable is matched with 'value-2' then statements2 are executed and the break statement transfers the program control out of the switch. If the match is not found, the 'default' block is executed. Remember, statements-x are executed in any case. Generally, switch case are executed for menu selection problems.

For example:

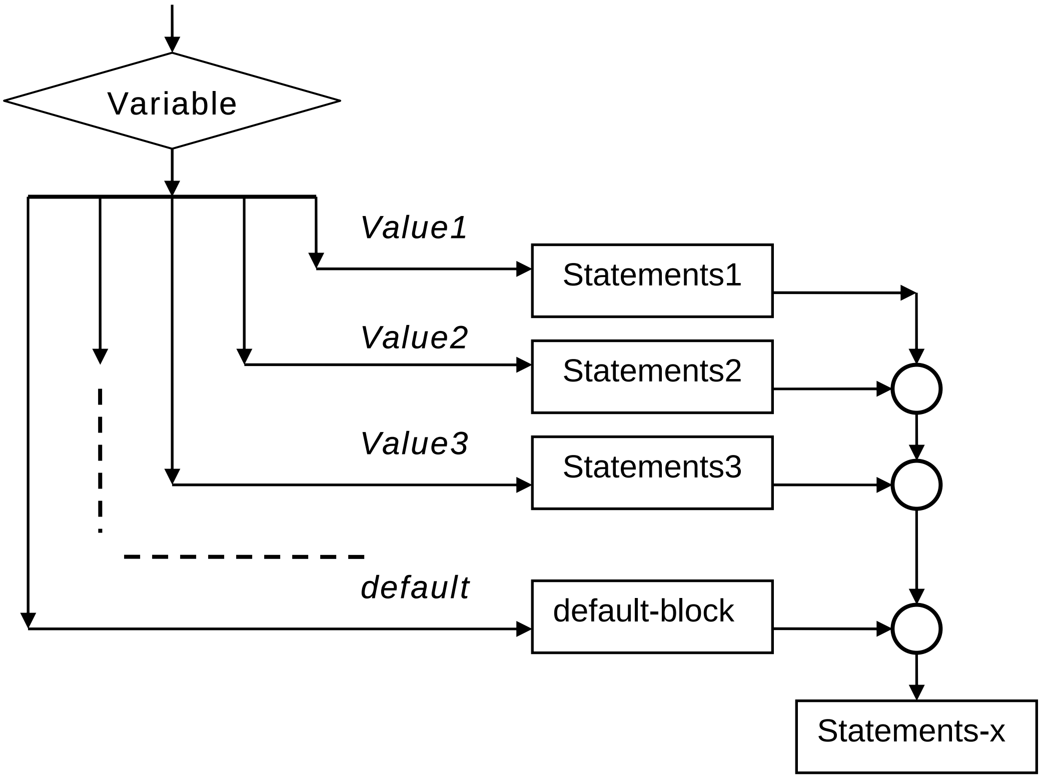
```
      /* Example 2.6 */
      int number = 2;
      switch(number)
      {
          case 0 :
                  printf("None\n");
                  break;
          case 1 :
                  printf("One\n");
                  break;
          case 2 :
                  printf("Two\n");
                  break;
          case 5 :
                  printf("Several\n");
                  break;
          default :
                  printf("Many\n");
                  break;
      }
```

Here, the output of this program will be:- Two

Flowchart: there are two ways in which a switch flowchart can be drawn.



OR



[Ref. 1i]

Rules of switch statement [Ref. 1j]

- The switch variable must be an integer or character type.
- Case labels must be constants of constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with a colon.
- The break statement transfers the program control out of the switch block. [Asked in W'07]

- The break statement is optional. When the break is not written in any 'case' then the statements following the next case are also executed until the 'break' is not found.
- The default case is optional. If present, it will be executed when the match with any 'case' is not found.
- There can be at most one default label.
- The default may be placed any where but generally written at the end. When placed at end it is not compulsory to write the 'break' for it.
- We can nest the switch statements.

Difference between if statement and switch statement

[Asked in S'07]

No.	If statement	Switch statement
1	It checks whether the condition is true or false	It is multi-way decision statement
2	It contains the condition which will evaluates to true or false	It contains the variable whose value is to be checked
3	It supports all types of conditions	It supports checking only integer and character values

4	Syntax:
	if(condition)
	{
	    //statements
	}
	else
	{
	    //statements
	}
		Syntax:
		switch(variable)
		{
		    case value1:
		        break;
		    case value2:
		        break;
		    default:
		}

The loop control structures [Ref. 1k, 2k]

Many times it is necessary to execute several statements repetitively for certain number of times. In such cases, we can use looping statements of C programming language. The statements are executed until a condition given is satisfied. Depending upon the position of the condition in the loop, the loop control structures are classified as the entry-controlled loop and exit-controlled loop. They are also called as pre-test and post-test loops respectively.

The 'while' loop [Ref. 1l, 2l]

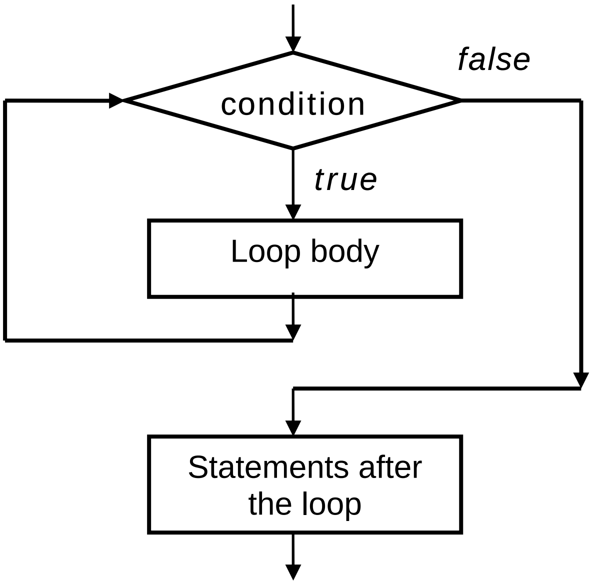
The 'while' is an entry-controlled loop statement having following general form:

```
while(condition)
```

```
    {
     //loop statements or body of loop
    }
```

Here, the condition is evaluated first, if it is true then loop statements or body of the loop is executed. After this the program control will transfer to condition again to check whether it is true or false. If true, again loop body is executed. This process will be continued until the condition becomes false. When is becomes false, the statements after the loop are executed. This is also called as pre-test loop.

Flowchart [Ref. 2m]



For example: If we want to print the number from 1 to 30 using a ‘while’ loop, it can be written as:

```
int num = 1;                  /* initialize counter */
while(num<=30)                /* check the condition */
{                             /* start of loop */
     printf(“\n%d”,num); /* print the number */
      num++;                  /* increment the counter */
}                             /* end of loop */
```

See one more practical example of ‘while’ loop. That is, if we want to find the addition of all the numbers from 10 to 20. It can be written as:

```
int x = 10, sum = 0;              /* initialize variables */
while(x <=20)                     /* check the condition */
{                                 /* start of the loop */
      sum = sum + x;              /* add the values */
      x++;                        /* increment counter */
}                                 /* end of the loop */
printf(“\nAddition is : %d”,sum);  /* print addition */
```

The do-while loop [Ref. 1m, 2n]

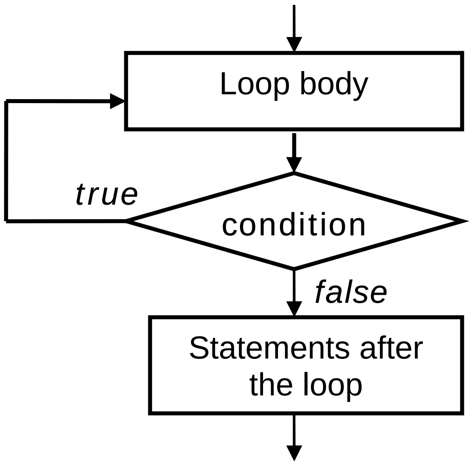
[Asked in W'08]

This is exit-controlled loop statement in which the condition of the loop statement is written at the end of the loop. It takes the following general form:

```
do
{
        //loop statements or loop body
}
while(condition);
```

Here, when the program control is reached at do statement the loop body or the statements inside the loop are executed first. Then at the end the condition is the 'while' is checked. If it is true, then program control again transfers to execute the loop statements. This process continues until the condition becomes false. In short, we can say that the loop statements are executed at least once without checking condition for its trueness.

Flowchart [Ref. 2o]



This loop is also referred as post-test loop structure. Consider the same examples used above: If we want to print the number from 1 to 30 using a 'do-while' loop, it can be written as:

```
int num = 1;                  /* initialize counter */
do                            /* do-statement */
{                             /* start of loop */
      printf("\n%d",num);     /* print the number */
      num++;                  /* increment the counter */
}                             /* end of loop */
while(num<=30);               /* condition */
```

See one more practical example of 'do-while' loop. That is, if we want to find the addition of all the numbers from 10 to 20. It can be written as:

```
int x = 10, sum = 0;               /* initialize variables */
```

```
do                                  /* do-statement */
{                                   /* start of the loop */
      sum = sum + x;                /* add the values */
      x++;                          /* increment counter */
}                                   /* end of the loop */
while(x <=20);                      /* check the condition */
printf(“\nAddition is : %d”,sum);   /* print addition */
```

Depending upon the problem statement given we can use either ‘while’ or ‘do-while’ loop statements.

Difference between ‘while’ and ‘do-while’
[Asked in S’07, W’07, S’08, W’08]

No.	‘while’ loop	‘do-while’ loop
1	It checks the condition at the start of the loop	It checks the condition at the end of the loop
2	This is of type entry controlled loop structure	This is of type exit controlled loop structure
3	The while loop is called as pre-test loop	The do-while loop is called as post-test loop
4	It is not guaranteed that how many times the loop body will get executed	The loop body will be executed at least once.

5	Syntax:
	`while(condition)`
	`{`
	`     //loop body`
	`}`
		Syntax:
		`do`
		`{`
		`     //loop body`
		`}while(condition);`

The ‘for’ loop [Ref. 1n, 2p]

The ‘for loop is an entry controlled loop structure which provides more concise loop structure having following general form:

```
for(initialization ; condition ; increment/decrement)
{
 //loop statements or loop body
}
```

The execution of ‘for’ loop statement takes as follows:

1. Initialization of loop control variables is done first using assignment operators such as: i = 1 or count = 0 etc. Remember this part is executed only once.
2. The condition given afterwards is checked then. If this condition is true, the statements inside the loop are executed. Else program control will get transferred nest statement after the loop. The

condition can be combination of relational as well as logical operators such as:
count < 10

3. When the body of the loop is executed, program control will get transferred back to 'for' statement for executing the third statement that is, 'increment/decrement'. In this part the loop control variable's value is incremented or decremented. Then program controls the condition again to check whether it is true or not. If true, the loop body executed again. And the process is continued again until the condition evaluates to false.

Consider the following code of the program:

```
int x;
for(x = 0; x < 10; x++)
{
     printf("%d", x);
}
```

This for loop is executed for 10 times and prints the 0 to 9 digits in one line. In short the statements are executed in following sequence:

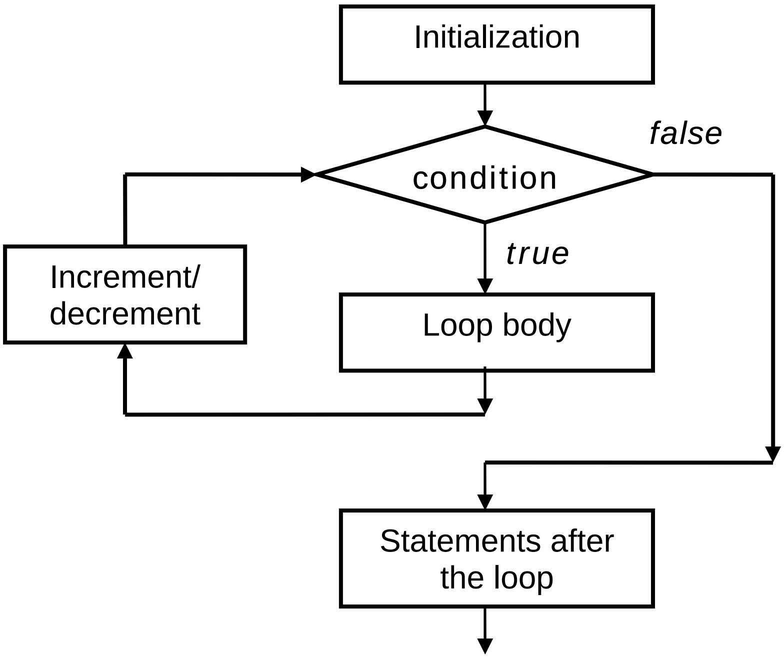
1. x = 0;
2. if(x<10)
   print value of x on screen
3. increment value of x by one
4. if(x<10)
   print value of x on screen
5. increment value of x by one

……….
When the condition becomes false, the loop gets terminated.

For example if we want to print all the even numbers from 1 to 25 it can be written using 'for' loop as:

```
int i;
for(i = 1;i < 25; i++)
{
     if(i%2 == 0)
        printf("\n%d", i);
}
```

The flowchart [Ref. 2q] of for loop can be shown as below:



Program to find the factorial of the number using ‘for’ loop

```
/* Example 2.7 */
#include<stdio.h>
main()
{
   int fact, num;
   printf("Enter the number: ");
   scanf("%d", &num);
   for(fact=1 ; num>0 ; num--)
     fact = fact * num;
   printf("Factorial: %d", fact);
}
```

Additional features of ‘for’ loop [Ref. 1o]

The ‘for’ loop have lot of capabilities that are not easily found in other loop constructs. These are listed below:

a. More than one variable can be initialized at a time in the ‘for’ statement. For example:

```
p = 1;
for(n = 0;n<10;n++)
```

can be re-written as–

```
for(p=1,n=0;n<10;n++)
```

b. Like initialization section, the increment section can also contain more than one increment or decrement statements.
For example:

```
for(n=1, m=50 ; n<=0 ; n++, m--)
```

c. The condition may have any compound relation and the testing need not be only loop control variable such as–

```
sum=0;
for(i=1; i<20 && sum<100; i++)
{
     sum = sum + i;
}
```

d. It is also permissible to use expressions in initialization and increment/ decrement sections such as–

```
for(x= (m+n)/2; x>0; x= x/2)
```

e. One or more sections can be omitted, if necessary. Such as–

```
m = 5;
for( ; m!=100 ; )
{
     printf(“%d\n”, m);
     m = m + 5;
}
```

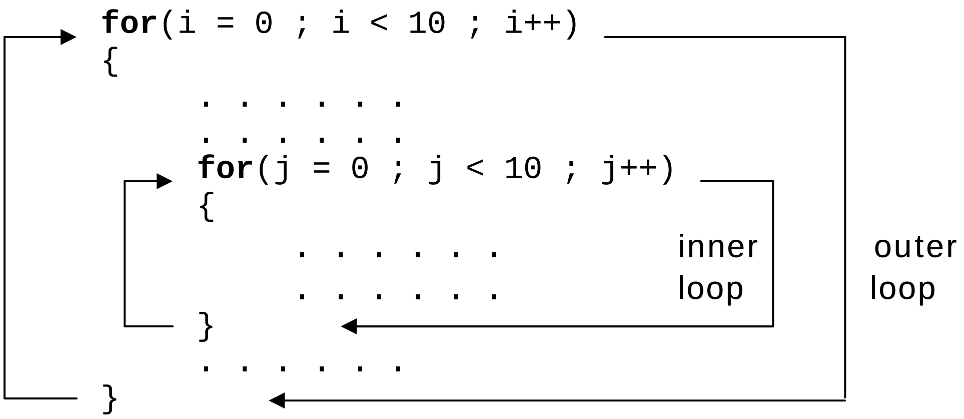
f. Giving a semicolon at the end of ‘for’ will not give any syntactical error but, all the iterations of the loop will be taken place. (Note: iteration is one execution of the loop). Such as–

```
for(i=0;i<100;i++);
```

This loop will be executed for 100 times without any output.

Nesting of the loops [Ref. 1p, 2r]

Nesting of the loop statements is allowed in C programming language. That is, we can write a ‘for’ loop inside another ‘for’ loop or any other loop in another loop statement. It can be written as follows:



```
for(i = 0 ; i < 10 ; i++)
{
     . . . . . .
     . . . . . .
     for(j = 0 ; j < 10 ; j++)
     {
          . . . . . .
          . . . . . .
     }
     . . . . . .
}
```

The nesting may be continued up to any desired level. Here, for each iteration of outer loop inner loop completes its all iterations. In above example, the inner loop statements will be executed for 10 X 10 = 100 number of times. See one application of this concept. For example we require following output:

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

In this case, the loops must be nested as shown below:

```
/* Example 2.8 */
#include<stdio.h>
main()
{
   int x,y;
   for(x = 0 ; x <= 3 ; x++)
   {
     for( y = 0 ; y <= x ; y++)
       printf("*");
     printf("\n");
   }
}
```

Jumps in the loop [Ref. 1q]

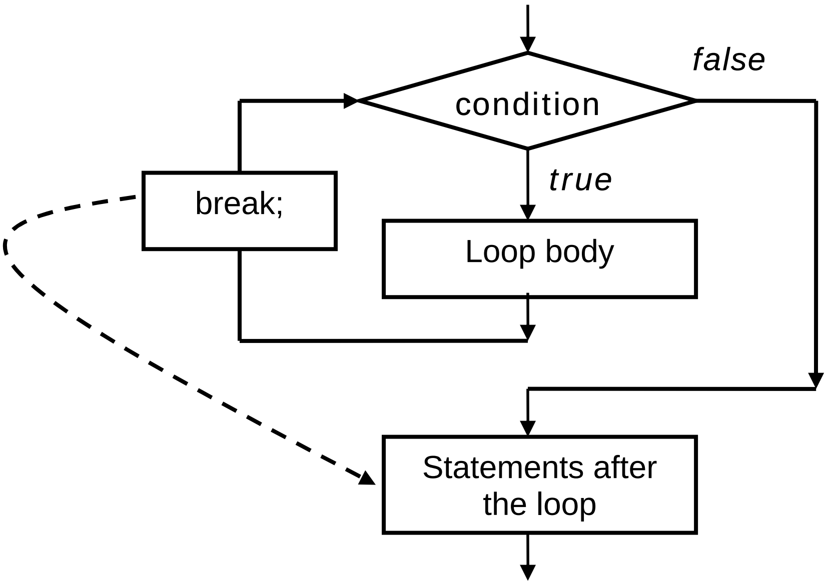
The break statement [Ref. 1r, 2s]

[Asked in W’08]

As we have already seen, the ‘break’ statement is used to exit out of the switch statement. It is having another useful application in loop control structures. That is, the 'break' statement is used to jump out of the loop. When the 'break' is executed inside the loop, the program control will not execute the loop for further iterations. Generally, the 'break' is associated with if-statement. General form of using 'break' is:

```
break;
```

Flowchart



The dotted line shows the path where the program control will transfer. For example:

```
/* Example 2.9 */
int x;
for(x = 10;x < 20; x++)
{
    if(x%4==0)
        break;
    printf("%d\n");
}
```

The output of this program will be:

```
10
11
```

Because, when value of x becomes 12, the condition will be evaluated to true and **break** get executed.

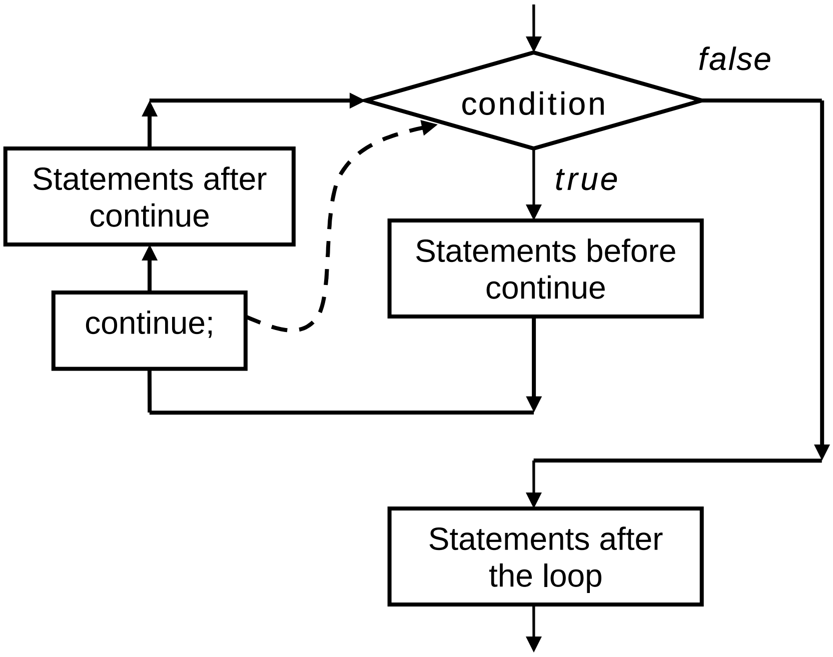
The 'continue' statement [Ref. 1s, 2t]

[Asked in S'07, W'07, S'08]

The continue statement is used to continue the next iteration of the loop by skipping the statements in between. That is, when the 'continue' is executed the statements following is will not be executed, the program control will directly transfer to next iteration of the loop by increment or decrement. Generally, the 'continue' is associated with if-statement. General form of using 'continue' is–

```
continue;
```

Flowchart:


false
condition
true
Statements after continue
Statements before continue
continue;
Statements after the loop

The dotted line shows the path where the program control will transfer. For example:

```
/* Example 2.10 */
int x;
for(x = 10;x < 20; x++)
{
     if(x%4==0)
          continue;
     printf("%d\n");
}
```

The output of this program will be:

```
10
11
13
14
15
17
18
19
```

It has skipped the numbers which are divisible by 4, for which we have skipped the printf statement.

Review Questions

Summer 2007: **Marks = 36**

1. State the use of ‘continue’ statement. 2
2. Describe if-else statement. 2
3. Write a program to display even numbers from 1 to n. Where n is accepted from user. 4
4. In what way does switch statement differ from an if statement 4
5. The following is the program segment:

```
x = 1;
y = 1;
if(n > 0)
    x = x + 1;
else
    y = y – 1;
printf(“%d %d”,x, y);
```

   What will be the values of x and y if n assumes a value of 1 and if n=0? 4
6. Write a program in ‘C’ to reverse the entered integer number as input through keyboard. 4
7. Compare while and do-while loop. 4
8. Explain nested if-else statement execution with its flowchart. 4
9. Write a program to print factorial of a number (W-07) 4
10. Write a program to find whether a given number is prime or not. 4

Winter 2007: **Marks = 36**

1. State difference between while and do-while loop 2
2. Write syntax for nested if-else statement. 2
3. Give syntax and example of switch-case statement 4
4. Explain importance of ‘break’ statement with switch-case. Give proper example. 4
5. Write a program to enter values of a and b variables. Calculate addition, division, multiplication, subtraction of a and b and display results. 4
6. Explain ‘continue’ statement with example. 4
7. Debug following program and write only errors if any:

```
#include<stdio.h>
main( )
{
    char X;
    while(X=0;X<=255;X++)
    printf(“\n character is %c”, X);
}
```

8. Write a program to display series of 11 to 30 numbers in reverse order as 30,29,28,……..2,1. Also calculate sum of 11 to 30 numbers and display it. 4
9. Write a menu driven program for the following options: 4
   i. To find whether entered no. is even or odd.

ii. To display sum of two entered numbers.

Summer 2008: **Marks = 32**

a) What is difference between while and do-while? 2
b) Describe nested if-else statement. 2
c) Write a program to display table up to 30. Use for loop. 4
d) Give any four forms of nested if-else statement. 4
e) Explain switch statement with suitable example. 4
f) Using while loop, write a program to print all Armstrong numbers between 1 and 500. If sum of cubes of each digit of the numbers is equal to the number itself, then number is called Armstrong number. For example: 153 = (1*1*1) + (5*5*5) + (3*3*3) 4
g) State use of continue statement over break statement using suitable example. 4
h) Write a program to print first 10 odd numbers using while loop. 4
i) Write a program which will print largest of three numbers using if-else. 4

Winter 2008: **Marks = 32**

a) State the use of break statement. 2
b) Explain conditional operator. 2
c) Write a program to display the sum of all even numbers from one to 'n'. Accept 'n' from user. 4
d) Explain else – if ladder with example. 4
e) Explain switch statement with flowchart. 4
f) Explain the working of do – while loop with example. 4
g) Differentiate between do-while and while loop. 4
h) Write a program to find sum of digits of five digit integer number. 4
i) Write a program to find out largest number among 3 integer nos. 4

Programming Exercises

1. Write a program to input two numbers from keyboard and find the result of subtraction of greater number – smaller number.
2. Input a year from keyboard and determine whether it is leap year or not using the conditional operator.
3. Accept three numbers from user and determine which of the two numbers or all three numbers are equal or not using logical operators.
4. Enter the amount saved in the bank from keyboard to a program. The interest rate to this amount is 5% per six months. Find out how much interest will be obtained on the amount saved for 1.5 years.
5. Imran's basic salary is input through the keyboard. His dearness allowance is 40% of basic salary, and house rent allowance is 20% of basic salary. Write a program to calculate his gross salary.

6. Input a number from user and determine whether it is fully divisible by 2 and 5 both or not?
7. Write a program to input a single character as input from user and determine whether it is vowel or not using else – if ladder.
8. Accept the height, base and hypotenuse values of right angled triangle from user and determine whether these values are satisfying the criteria being the triangle as right angled. (Hint: use Pythagoras theorem).
9. Input the marks of five different subjects CMS, CPR, ELC, EMT, and ETE from user and determine whether the student is fully passed in all subjects or not. If yes calculate the percentage obtained. (Use maximum marks of paper as 100).
10. Accept a four digit number from keyboard and determine whether all four digits of that number are same or not.
11. Write a program to input the year from user and find how many days will be there on this year in the month of February. Use nested if-else?
12. Input two numbers from keyboard and determine whether their multiplication is negative or not? If no check whether it is odd or even?
13. Create a menu driven program to input a character from user and determine:
    i. Whether it is alphabet or not.
    ii. Whether it is in upper – case or not.
14. Create a menu driven program to input a number from user for following options:
    i. To find the last digit of that number.
    ii. To find whether it is positive or not.
15. Create a menu driven program to input two numbers from user for following options:
    i. To find larger number between them.
    ii. To find remainder in their division.
16. Create a menu driven program to input a float number from user for following options:
    i. To find cube of that number.
    ii. To find square root of that number.
17. Create a menu driven program to input a character from user for following options:
    i. To determine whether it is vowel or not.
    ii. To display the character next to it alphabetically.
18. Create a menu driven program to input total marks of one student from user for following options:
    i. To check whether he/she has passed the exam or not.
    ii. To find how much percentage obtained by him/her.

    (Consider marks are aggregated by 5 subjects.)
19. Write a program to print following output using for loop:

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

20. Accept a number from user and print it in reverse order using do-while loop.
21. Write a program to find sum of all even numbers from n to m. Accept values of variables n and m from keyboard.
22. Generate the following Fibonacci series using while loop till 'n' number. Accept n from user.

    0 1 1 2 3 5 8 13 … … … n
23. Find the addition of all the numbers from n to –n. Take value of n as input from keyboard. Use for loop.
24. Using while loop, write a program to print all Armstrong numbers between 1 and 500. If sum of cubes of each digit of the numbers is equal to the number itself, then number is called Armstrong number. For example: 153 = (1*1*1) + (5*5*5) + (3*3*3)

References

1. **Programming in ANSI C**
   by, E Balagurusamy, Tata McGraw Hill, 4th Edition
   a. Chapter 5, Topic 5.2, Page No. 114
   b. Chapter 5, Figure 5.2, Page 116
   c. Chapter 5, Topic 5.4, Page 119
   d. Chapter 5, Figure 5.5, Page 120
   e. Chapter 5, Topic 5.5, Page 122
   f. Chapter 5, Topic 5.6, Page 126
   g. Chapter 5, Figure 5.9, Page 127
   h. Chapter 5, Topic 5.7, Page 129
   i. Chapter 5, Figure 5.11, Page 131
   j. Chapter 5, Page 133
   k. Chapter 6, Topic 6.1, Page 152
   l. Chapter 6, Topic 6.2, Page 154
   m. Chapter 6, Topic 6.3, Page 157
   n. Chapter 6, Topic 6.4, Page 159
   o. Chapter 6, Page 161
   p. Chapter 6, Page 164
   q. Chapter 6, Topic 6.5, Page 166
   r. Chapter 6, Topic 6.5, Page 167
   s. Chapter 6, Page 171

2. **Let us C**
   By, Yashwant Kanetkar, BPB Publications, 7th Edition
   a. Chapter 2, Page 51
   b. Chapter 2, Page 56
   c. Chapter 2, Page 58
   d. Chapter 2, Page 59
   e. Chapter 2, Figure 2.5, Page 60
   f. Chapter 2, Page 61
   g. Chapter 2, Page 62
   h. Chapter 2, Page 64
   i. Chapter 2, Page 67
   j. Chapter 4, Page 136
   k. Chapter 3, Page 98
   l. Chapter 3, Page 99
   m. Chapter 3, Figure 3.2, Page 101
   n. Chapter 3, Page 121
   o. Chapter 3, Figure 3.5, Page 121
   p. Chapter 3, Page 108
   q. Chapter 3, Figure 3.4, Page 111
   r. Chapter 3, Page 114
   s. Chapter 3, Page 118
   t. Chapter 3, Page 120

Chapter 03

Arrays and Strings

Lectures allotted: **14**

Marks Given: **12**

Contents:

3.1 Arrays

Declaration and initialization of one dimensional array,

Two dimensional and character arrays,

Accessing array elements

3.2 Declaration and initialization of string variables,

String handling functions from standard library:

strlen( ), strcpy( ), strcat( ), strcmp( )

Array [Ref. 1a]

[Asked in S'08]

An array is a fixed-size collection of elements having same data-type. It is simply a grouping of like-type data. Array is one of the derived data-type (derived from other types of the data). That is array can be used to represent a list of numbers, a list of names etc.

One Dimensional Array [Ref. 1b]

A list of items can be given one variable name using only one subscript; such a variable is called as one dimensional array. It is very generalized form of the array.

Declaration of one dimensional array [Ref. 1c]

[Asked in W'08]

Like other general variables, the array is also declared in the program before its use. The C programming language follows following syntax to declare an array as –

```
data-type variable-name[size];
```

[Asked in S'07]

The 'data-type' specifies type of the array. That is, int, char, float etc. The 'variable-name' is the name given to the array, by which we will refer it. Finally, size is the maximum number of elements that an array can hold.

For example:

```
int var[10];
```

This declares the array 'var' containing 10 different integer elements. We can store 10 integer constants in the variable 'var'.

Initialization of one dimensional array [Ref. 1d]

[Asked in W'08]

After an array is declared, it must be initialized. Otherwise they will contain the garbage values. There are two different ways in which we can initialize the array:

1. Compile time
2. Run time

Compile time initialization

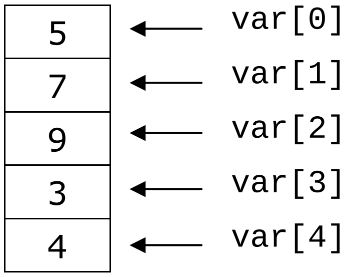
This initialization is done while writing the program itself. Following is the syntax of initialization of the array.

```
data-type array-name[size] = { list of values };
```

For example:

```
int var[5] = {5, 7, 9, 3, 4};
```

Here, the array variables are initialized with the values given in front of it. That is, the 0th element of array will contain 5, 1st will contain 7 and so on. Remember, the array indexing or subscripting is always done from 0th position. The values of the array will be get stored as following manner



Each element in the array will be referred individually by its index such as, var[0], var[1], var[2], . . .

They can also be initialized individually such as:

```
var[0] = 5;
var[1] = 7;
var[2] = 9;
```

etc. after declaration of the array.

In the array declaration, the size can also be omitted. In such cases the compiler will automatically identify the total number of elements and size will be given to it. For example:

```
int max[ ] = {8, 6, 7, 4, 5, 3, 0, 1};
```

This array is not declared with size, but initialized with values. Here, size of the array will be automatically set to 8. Now it can be referred as the general array onwards.

Compile time initialization can also be done partially. Such as –

```
float x[5] = {1.2, 6.9, 4.1};
```

Here, array 'x' is having size 5, but only 0th, 1st, and 2nd elements are declared with values. The remaining 3rd and 4th element of the array will be automatically set to 0. This property is applicable to all numerical type of the array in C. remember, however, if we have more values than the declared size, the compiler will give an error. For example –

```
int arr[5] = {56, 10, 30, 74, 56, 32};
```

This is the illegal statement in C.

Run time initialization

An array can initialized at run time by the program or by taking the input from the keyboard. Generally, the large arrays are declared at run time in the program it self. Such as –

```
int sum[20];
for (i = 0;i<20;i++)
     sum[i] = 1;
```

Here, all the elements of the array 'sum' are initialized with the value 1. Remember the loop control structures are applicable in these cases.

We can also use the scanf function to input the values from the user. In such cases, the loop control structure is applicable. For example:

```
int sum[10], x;
printf("Enter 10 numbers: ");
for(x=0;x<10;x++)
     scanf("%d",&sum[x]);
```

Here, the array 'sum' is initialized by taking the values as input from the keyboard.

```
/* Program 3.1 */
/* Input 10 numbers from user and find the total of all of
them */
#include<stdio.h>
main( )
{
     int val[10], i, total=0;
     printf("Enter any ten numbers: ");
     for(i=0;i<10;i++)
         scanf("%d", &val[i]); // input numbers

     for(i=0;i<10;i++)
         total = total + val[i];  // find total

     printf("\nTotal is: %d", total);
}
```

Sample Output:

```
Enter any ten numbers: 1 2 3 4 5 6 7 8 9 10
Total is: 55
```

Find maximum number from the array of 10 float values.

Algorithm:

1. Start

```
2.    Initialize the array of 10 elements.
3.    Let i = 1,
4.    Consider max = 0th element of the array.
5.    If ith  element of array > max
      then max = ith  element of array
6.    Increment value of i by 1.
7.    If i ≠ 10 go to step 5.
8.    Print the value of max.
9.    Stop.
```

Two dimensional arrays [Ref. 1e]

[Asked in S'07, W'07, S'08, W'08]

It is also called as array of arrays. Many times it is required to manipulate the data in table format or in matrix format which contains rows and columns. In these cases, we have to give two dimensions to the array. That is, a table of 5 rows and 4 columns can be referred as,

```
table[5][4]
```

The first dimension 5 is number of rows and second dimension 4 is number of columns. In order to create such two dimensional arrays in C, following syntax should be followed.

```
datatype arrayname[rows][columns];
```

For example:

```
int table[5][4];
```

This will create total 20 storage locations for two dimensional arrays as,

	Columns			
Rows	[0][0]	[0][1]	[0][2]	[0][3]
	[1][0]	[1][1]	[1][2]	[1][3]
	[2][0]	[2][1]	[2][2]	[2][3]
	[3][0]	[3][1]	[3][2]	[3][3]
	[4][0]	[4][1]	[4][2]	[4][3]

We can store the values in each of these memory locations by referring there respective row and column number as,

```
table[2][3] = 10;
table[1][1] = -52;
```

Like one dimensional arrays, two dimensional arrays can also be initialized at compile time as,

```
    int table[2][2] = {8, 5, 9, 6};
```

It will initialize the array as shown below, [Ref. 1f]

8	5
9	6

Following declaration and initialization is more understandable than previous one.

```
    int table[][] = {{8, 5}, {9, 6}};
```

or

```
    int table[][] = {
                        {8, 5},
                        {9, 6}
                    };
```

The large two dimensional arrays can also be initialized using loops such as,

```
int arr[5][5];
int z = 0, x, y ;
for(x=0;x<5;x++)
{
   for(y=0;y<5;y++)
   {
     arr[x][y] = z;
     z++;
   }
}
```

This will initialize the array ‘arr’ as,

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Same can be printed using nesting of two loops as,

```
for(x=0 ; x<5 ; x++)
   for(y=0 ; y<5 ; y++)
       printf(”%d”, arr[x][y]);
```

Basically, the two dimensional array are used for manipulating different matrix operations. Program 3.2 shows the application of two dimensional arrays.

```
//Addition of matrices Program 3.2
#include<stdio.h>
main()
{
         int x[][] = { {8,5,6},
                       {1,2,1},
                       {0,8,7}
                     };

         int y[][] = { {4,3,2},
                       {3,6,4},
                       {0,0,0}
                     };
        int i,j;
         printf("First matrix: ");
         for(i=0;i<3;i++)
         {
            for(j=0;j<3;j++)
              printf("%d", x[i][j]);
            printf("\n");
         }
         printf("Second matrix: ");
         for(i=0;i<3;i++)
         {
            for(int j=0;j<3;j++)
              printf("%d", y[i][j]);
            printf("\n");
         }
         printf("Addition: ");
         for(i=0;i<3;i++)
         {
            for(int j=0;j<3;j++)
              printf("%d", x[i][j]+y[i][j]);
            printf("\n");
         }

      }
  }
```

Output:

```
First matrix:
 8 5 6
 1 2 1
 0 8 7
```

```
Second matrix:
 4 3 2
 3 6 4
 0 0 0
Addition:
 12 8 8
 4 8 5
 0 8 7
```

Strings [Ref. 1g]

[Asked in W'07]

The string is sequence of characters that is treated as a single data item. In general, the character array is called as string. Any group of characters defined between double quotation marks is a string constant. Such as –

```
    "The C Programming Language."
```

Declaring a string [Ref. 1h]

C does not support string as the data type. However is allows us to represent strings as character arrays. It can be declared as –

```
    char string_name[size];
```

Here, string_name is the valid variable name given to the string and 'size' determines the number of characters in the string. For example:

```
    char city[10];
    char name[30];
```

Here, 'city' is the character array or a string can be stored with 10 characters. The 'name' will store 30 characters as well.

Initializing a string [Ref. 1h, 1i]

Like numerical arrays the strings can also be initialized when they are declared which is referred as compile time initialization. It can be done in following three forms:

```
    char city[6] = "Nashik";
    char city[ ] = "Nashik";
    char city[6] = {'N', 'a', 's', 'h', 'i', 'k' };
```

For initializing the string at run-time, we use scanf function with %s format specifier such as:

```
    char str[10];
    scanf("%s", str);
```

This statement will initialize the string 'str' by taking the input from the keyboard. This will read the character sequence until occurrence of first separator such as space character, tab or enter and store it into 'str'. We can also use the following function to input the string including spaces.

```
    char str[10];
    gets(str);
```

The gets() function defined in stdio.h file reads the string from the keyboard until the enter is not pressed.

For printing the string, we can use the printf function in the similar format such as:

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

Arithmetic operations on character [Ref. 1j]

C allows us to manipulate the character as the same way we do for the integers. Every character is associated with an integer value with it. This value is called as ASCII value. If we add two characters in each other, the ASCII values of them are added in each other. This concept is most useful in solving most of the problems in C. for example, if we want to print the ASCII value or integer value of character 'm' on screen we can use:

```
    printf("%d", 'm');
```

or

```
    char ch = 'm';
    printf("%d",ch);
```

If we want to change value of any character variable to its next subsequent character value, we can write as:

```
    ch = ch + 1;
```

```
/* Program 3.3: to count the length of the string */
#include<stdio.h>                         [Asked in W'08]
main()
{
    char str[20];
    int x;
    printf("Enter the string: ");
    scanf("%s", str);
    for(x=0;str[x]!='\0';x++);

    printf("\nLength of the string is: %d",x);
}
```

```
/* Program 3.4: to count the upper-case characters from the
string */
#include<stdio.h>
main()
{
    char str[20];
    int x, upper=0;
    printf("Enter the string: ");
    scanf("%s", str);
    for(x=0;str[x]!='\0';x++)
    {
        if(str[i]>='A' && str[i]<='Z')
            upper++;
    }
    printf("\nUpper case characters are: %d",upper);
}
```

String handling functions [Ref. 1k]

C library is supporting a large collection of string handling functions. They all are defined in string.h file.

1. strlen()

[Asked in W'07, W'08]

This function counts and returns number of characters from the string. It takes the following form:

```
n = strlen(string);
```

here, n is an integer variable, which receives value of length of the string. The argument to this function may be a string constant or string variable. The counting ends at the first null character.
For example:

```
char name[10] = "Yellow";
int len;
len = strlen(name);    //'len' will be 6.
```

2. strcat()

[Asked in W'07, S'08]

This function joins two strings together. That is, this function is used to concatenate the strings. It takes the following general form:

```
strcat(string1, string2);
```

here, string1 and string2 are character arrays. When this function is executed, content of string2 is attached at the end of string1. Content of string2 remains unchanged.
For example:

```
char str1[ ] = “New”;
char str2[ ] = “Delhi”;
strcat(str1, str2);
printf(“%s”, str1);    //outputs “NewDelhi”
printf(“%s”, str2);    //outputs “Delhi”
```

We can concatenate / join a string constant to the string variable such as:

```
strcat(str1, “York”);
printf(“%s”, str1);    //outputs “NewYork”
```

3. strcpy()

[Asked in S’07, W’07, W’08]

This function almost works like string-assignment operator. That is, strcpy() function is used to copy contents of one string into another. It takes the following general form:

```
strcpy(string1, string2);
```

This will copy contents of string1 into string2.
For example:

```
char city1[5] = “Pune”;
char city2[5];
strcpy(city2, city1);
printf(“%s”, city1);   //outputs “Pune”
printf(“%s”, city2);   //outputs “Pune”
```

We can also use the constant as the parameter to the strcpy(). Such as:

```
strcpy(city2, “Pimpri”);
```

4. strcmp()

[Asked in S’07, S’08]

This function is used to compare two strings and accordingly returns the value. That is, it returns the numerical difference between the strings. It takes the following general form:

```
strcmp(string1, string2);
```

Here, string1 and string2 may be string variables or strings constants. The function will compare the strings and returns the value. if this value of 0, implies that both the strings are equal. If it is positive, implies that string1 is greater and if negative, implies that string2 is greater. For example:

```
strcmp(city1, city2);
strcmp("Pune", city1); etc.
```

This function is case-sensitive function. The case-insensitive form of this function is also available given below:

```
strcmpi(string1, string2);  or
stricmp(string1, string2);
```

5. strrev()

This function reverses the contents of the string. It takes the following general form:

```
strrev(string);
```

Here, the 'string' is the string *variable* whose value is to be reversed. For example:

```
char str[10] = "Sakeena";
strrev(str);
printf("%s", str);     //print aneekaS on the screen.
```

```
/* Program 3.5: Use of string handling functions */
#include<stdio.h>
#include<string.h>
main()
{
    char str1[10], str2[10];
    printf("\nEnter first string: ");
    scanf("%s", str1);
    printf("\nEnter second string: ");
    scanf("%s", str2);

    printf("\nLength of second string: %d", strlen(str2));

    printf("\nReverse of first string: %s", strrev(str1));

    if(strcmp(str1, str2)==0)
        printf("\nStrings are equal");
    else
        printf("\nStrings are not equal");
```

```
     printf(“Joined string: %s”, strcat(str1, str2));
     strcpy(str1,str2);
     printf(“\nCopied second string in first: %s”, str1);

}
```

Array of Strings [Ref. 1l]

We can also create the array of the string. It is basically a two-dimensional array of characters and declared in the same way as that of general two-dimensional array given below:

```
     char tab[5][10] = {     “Thomas”,
                             “Rajneesh”,
                             “Atish”,
                             “Aneeta”,
                             “Nilofar”
                       };
```

In above example, 5 specify total number of strings that the array may contain and 10 specifies maximum length of each string in the array. This will create the following table in memory.

T	h	o	m	a	s	\0			
R	a	j	n	e	e	s	h	\0	
A	t	i	s	h	\0				
A	n	e	e	t	a	\0			
N	i	l	o	f	a	r	\0		

We can access these elements using the array in the following manner.

```
     for(x=0;x<5;x++)
     printf(“\n%s”, tab[x]);
```

For initializing the array of strings, we can also use the same way but using scanf function.

```
/* Program 3.6: Print reverse of 5 different strings */
#include<string.h>
#include<stdio.h>
main()
{
     char str[10][10];
     int i;
     printf(“\nEnter 10 strings: ”);
     for(i=0;i<10;i++)
          scanf(“%d”,str[i]);
     for(i=0;i<10;i++)
          scanf(“%d”,strrev(str[i]));
}
```

Review Questions

Summer 2007: **Marks = 16**

- **a)** Define and declare two dimensional array. 2
- **b)** State use and syntax of strcmp ( ) function. 2
- **c)** Explain the need for array variables. 4
- **d)** Explain strcpy () function in detail. 4
- **e)** Write a program to accept a string and display a list of ASCII codes which represent accepted string. 4

Winter 2007: **Marks = 16**

- **a)** What do you mean by character array? 2
- **b)** Give syntax for strcat ( ) string function. 2
- **c)** Explain what is two dimensional array. Also explain how to declare and initialize two dimensional array with example. 4
- **d)** Explain strlen ( ) and strcmp ( ) string functions with example. 4
- **e)** Write a program to find the largest number from given array. 4

Summer 2008: **Marks = 16**

- **a)** What is array? How it is declared? 2
- **b)** State use and syntax of strcat( ) function. 2
- **c)** Explain representation of two dimensional array with example. 4
- **d)** Explain strcmp( ) function in detail. 4
- **e)** Write a program to copy contents of one array into another array. 4

Winter 2008: **Marks = 16**

- **a)** Declare and initialize the one dimensional integer array with 10 elements. 2
- **b)** State the use and syntax of strlen( ) function. 2
- **c)** Describe a method of declaring and initialization of two dimensional array with example. 4
- **d)** Explain strcpy( ) function. 4
- **e)** Write a program to accept a string and display the length of it without using standard library function. 4

Programming Exercises

1. Enter ages of ten different men from user and find out who is the eldest in all?
2. Accept 10 different values from keyboard; determine their average and find out is there any number in the array whose value equals to average.
3. Write a program to copy contents of one array into another array. Accept size of both arrays as 7.
4. Input 10 different numbers from user. Count and display total even numbers from it.

5. Accept values of two different arrays (size 5) from user and determine how many numbers they are having common.
6. Write a program to accept 15 different float number from user and display those numbers which are positioned on odd index positions in reverse order. For example, you have to print the numbers on array[13], array[11], array[9], array[7]………so on.
7. Initialize the matrix of 3 X 3 from user and determine whether it is zero matrix or not?
8. Accept the values of 2 X 2 matrix from keyboard and find determinant of it.
9. Initialize a 3 X 3 matrix and copy the contents of it into a single dimensional array whose size is 9. Print all the values of 1D array.
10. Declare a 2D array of 4 X 3. Initialize first row values with 5, second row values with 9, third row values with 3, and fourth row values with 7 using loop control structures.
11. Accept a 3 X 5 matrix from keyboard and copy the contents of it into another 5 X 3 matrix by finding the transpose.
12. Input a 4 X 4 matrix and find the diagonal positions of it contains all values same or not?
13. Accept any sentence from keyboard using gets( ) function and count total number of words in it.
14. Accept any string from keyboard, convert it into the upper case and display it.
15. Read a string from keyboard and inverse the case of it. That is, convert lower case letters to upper-case and vice versa.
16. Input any two strings from user and find whether these are equal or not. Don't consider the case.
17. Read a string from user and analyze it. That is, count total number of alphabets, digits, special symbols and display their counts.
18. Write a program to accept a string and display a list of ASCII codes which represent accepted string (Asked in Summer 2007 Examination)

References

1. **Programming in ANSI C**
   by, E Balagurusamy, Tata McGraw Hill, 4th Edition
   a. Chapter 7, Topic 7.1, Page 190
   b. Chapter 7, Topic 7.2, Page 192
   c. Chapter 7, Topic 7.3, Page 193
   d. Chapter 7, Topic 7.4, Page 195
   e. Chapter 7, Topic 7.5, Page 199
   f. Chapter 7, Topic 7.6, Page 204
   g. Chapter 8, Topic 8.1, Page 229
   h. Chapter 8, Topic 8.2, Page 230
   i. Chapter 8, Topic 8.3, Page 231
   j. Chapter 8, Topic 8.5, Page 231
   k. Chapter 8, Topic 8.8, Page 244
   l. Chapter 8, Topic 8.9, Page 250

Chapter 04

Functions and Structures

Lectures allotted: **14**

Marks Given: **12**

Contents:

4.1 Functions

Need of functions,

Scope and lifetime of variables,

Defining functions,

Function calls (call by value, call by reference),

Return values,

Storage classes,

Category of function (No argument No return value, No argument with return value, argument with return value), recursion

4.2 Structures

Defining structure,

Declaring and accessing structure members,

Initialization of structure,

Arrays of structure.

Function

[Asked in W'07, W'08]

A function is self-contained block of statements that performs particular task. C functions can be classified into two types:

- Library functions
- User-defined functions

The library functions are the functions which are already defined in C's functions library i.e. header files. For example, the functions scanf() and printf() are the library functions defined in file stdio.h same as functions sqrt() is defined in math.h and getch() is defined in conio.h. User defined function is the function defined by the programmer who has written the program. The task to perform is decided by the user. For example, the main() function is an user-defined function. We decide what is to be written in this function.

Need of user-defined function

[Asked in W'08]

1. The functional program facilitates top-down modular programming approach as shown in figure below.

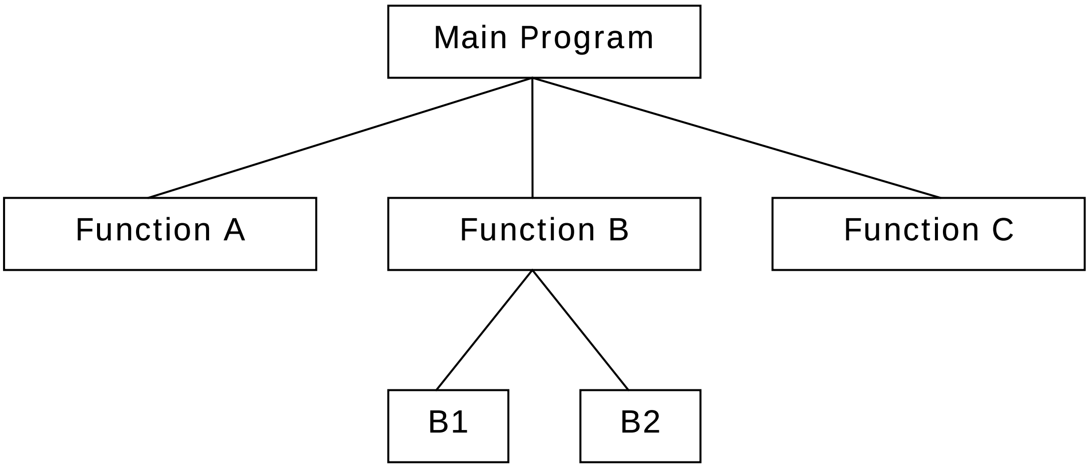


Fig. Top down modular programming approach

2. The length of source program can be reduced by using functions at appropriate places.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on what others have already done. That is we can reuse the functions already defined in some program files.

Elements of user-defined functions

There are some similarities exists between functions and variables in C.

1. Both function name and variable names are considered as identifiers and therefore they must follow the rules of creating variable's name.

2. Like variables, functions have data types associated with them.
3. Like variables, function names and their types must be declared and defined before they are used in the program.

In order to make use of user defined functions, we need to establish three elements that are related to functions.

1. Function definition
2. Function call
3. Function declaration

Definition of functions

[Asked in S'08]

The function definition is an independent program module that is specially written to implement the requirements of the function. So it is also called as function implementation. It includes following topics:

- Function Header
- Function Body

The general form of function definition is as given below:

```
function_type function_name(parameters list)
{
     local variables declaration;
     executable statement1;
     executable statement2;
     - - - - - -;
     - - - - - -;
     return statement;
}
```

The first line `function_type function_name(parameters list)` is known as function header and the statements enclosing the curly braces are known as function body.

Function Header

It includes three parts:

1. Function type
   It specifies type of the value that the function is expected to return to the program calling the function. It is also called as return type. If function is not returning any values we have to specify it as void.
2. Function name
   It is any valid C identifier name and therefore must follow the rules for creation of variable names in C.
3. Parameters list
   It declares variables that will receive the data sent by the calling program. They serve as input data to the function to carry out specific

task. Since, they represent the actual input values they are referred as formal parameters or formal arguments.

For example:

```
int findmax(int x, int y int x)
{
    --------
}

double power(double x, int n)
{
    --------
}

float quad(int a, int b, int c)
{
    --------
}

void print_line()
{
    --------
}
```

Function Body

It contains the declarations and statements necessary for performing required task. The body enclosed in braces, contains three parts:

1. Local variables declaration
   It specifies the variables needed by the function locally.
2. Function Statements
   That actually performs task of the function.
3. The return statement
   It returns the value specified by the function.

For example:

```
float mul(float x, float y)
{
    float result;          /* local variable */
    result = x * y;        /* find the result */
    return(result);        /* return the result */
}
void display(void)
{
    printf("Hello World!");     /* only print the value */
}
void sub(int a, int b)
```

```
{
    printf(“Subtraction: %d”, a – b);    /* no variables */
    return;                  /* optional */
}
```

Return values

A function may or may not send back any value to the calling function. If it does, it is done by return statement. The return statement also sends the program control back to the calling function. It is possible to pass a calling function any number of values but called function can only return one value per call, at most.

The return statement can take one of the following forms:

```
    return;
    return(value);
    return(variable);
    return(expression);
```

The first plain return does not return any value; it acts much as closing brace of the function. The remaining three statements can eliminates the brackets also. When the return is executed, the program control immediately transfers back to the calling function. None of the statements written after return are executed afterwards. For example:

```
    return(x);
    printf(“Bye…Bye”);
```

In this case the printf statement will not be executed in any case. Because the return statement will transfer program control immediately back to the calling function. Some examples of return are:

```
    int div(int x, int y)
    {
        int z;
        z = x / y;
        return z;
    }
```

Here, instead of writing three statements inside the function div(), we can write a single statement as,

```
    return(x*y);
    OR
    return x*y;
```

A function may have more than one return statement when it is associated with any condition such as,

```
if(x>y)
    return x;
else
    return y;
```

In this code, in any condition, only one return statement will be executed.

Function calls

A function can be called by simply using function name followed by a list of actual parameters (or arguments) if any, enclosed in parentheses. For example:

```
main()
{
    float m;
    m = mul(5.2, 3.71);    /* function call */
    printf("\n%d",m);
}
```

When compiler encounters the function call, it transfers program control to the function mul() by passing values 5.2 and 3.71 (actual parameters) to the formal parameters on the function mul(). The mul() will performs operations on these values and then returns the result. It will be stored in variable m. We are printing the value of variable m on the screen.

There are a number of different ways in which the function can be called as given below:

```
mul(5.2, 3.71);
mul(m, 3.71);
mul(5.2, m);
mul(m, n);
mul(m+5.1, 3.71);
mul(m+3.6, m+9.1);
mul(mul(6.1,8.2), 4.5);
```

Only we have to remember that the function call must satisfy type and number of arguments passed as parameters to the called function.

Function Prototype / Function Declaration

[Asked in S'07]

Like variables, all functions in C program must be declared, before they are used in calling function. This declaration of function is known as function prototype. It is having following syntax to use:

```
function_type function_name (parameter list);
```

this is very similar to function header except the terminating semicolon. For example, the function mul() will be declared as,

```
float mul(float, float);
```

Generally, prototype declarations as not necessary, if the function have been declared before it is used. A prototype declaration may be placed in two places in the program.

- Above all the functions
- Inside function definition

We place declaration above all the functions this prototype is known as global prototype. That is, we can use this function in any part on the program. When we place the function definition in the local declaration section of another function it is referred as local prototype.

Categories of the function

Depending upon whether arguments are present or not and whether value is returned or not functions are categorized as:
Category 1: Functions with no arguments and no return value
Category 2: Functions with arguments and no return value
Category 2: Functions with arguments and a return value
Category 2: Functions with no arguments and a return value

Functions with no arguments and no return value

For example:

```
void printline(void)
{
     int i;
     for(i=0;i<25;i++)
        printf(“–”);
}
```

Functions with arguments and no return value

For example:

```
void findmax(int a, int b)
{
     if(a>b)
          printf(“%d”, a);
     else
          printf(“%d”, b);
}
```

Functions with arguments and return value

For example:

```
int addition(int x, int y)
{
	int s;
	s = x + y;
	return (s);
}
```

Functions with arguments and a return value

For example:

```
float value_of_PI(void)
{
	float pi = 22/7;
	return pi;
}
```

Recursion

[Asked in W'08]

It is a special case of calling a function when a function calls itself again and again. It forms a chaining of calling a function. For example:

```
void main(void)
{
	printf("\nHello");
	main();			/* recursive call */
}
```

When executed, this program will display "Hello" on the screen indefinitely.

Another useful example of recursion is evaluation of factorial of a number. The factorial of number is expressed as a series of repetitive multiplications as, for example:

```
factorial of 5 = factorial of 4 × 5
factorial of 4 = factorial of 3 × 4
factorial of 3 = factorial of 2 × 3
factorial of 2 = factorial of 1 × 2
factorial of 1 = 1
```

This statement will be transferred into recursive process in function as given below:

```
int factorial(int n)
{
    int fact;
    if(n==1)
        return 1;
    else
        return(n*factorial(n-1));
}
```

Consider another example of finding value of X^y using recursion:

```
#include<stdio.h>
#include<conio.h>
int power(int x, int y)
{
    if(y<1)
        return(1);
    else
        return( x * power(x,--y));
  }
void main()
{
    printf("%d", power(4, 3));
    getch();
}
```

Scope, Lifetime and Visibility of variables

Scope is the region in which a variable is available for use. Depending upon this region, the scope is categorized into two types, that is local scope and global scope. For example:

```
int num1;     /* global scope */
main()
{
    int num2;/* local scope */
    ------
    ------
}
```

In this code, the variable num1 declared globally so it is available for use in the whole program. The variable num2 is declared locally, so it can be used only in main() (that is, in the function where it is declared).

The program's ability to access a variable from memory is called as visibility of variable. Whereas, the lifetime of the variable is duration of the time in which a variable exists in the memory during execution.

Rules of use

1. Scope of global variable is the entire program.
2. The scope of local variable begins at point of declaration and ends at the end of block or function in which it is declared.
3. The scope of a formal function is its own function.
4. Lifetime of a variable declared in main is the entire program execution time, although its scope is only main function.
5. All variables have visibility in their scope, provided they are not declared again.
6. If a variable is re-declared within its scope again, it loses its visibility in the scope of the re-declared variable.

Storage Classes

The properties of the variables tell us about following things, referred as storage class:

1. Where the variable would be stored?
2. What will be the initial value of the variable? (i. e. default initial value)
3. What is scope of the variable?
4. What is lifetime of the variable?

There are four different types of storage classes:

1. Automatic
2. Register
3. Static
4. External

1. Automatic Storage class

They are created when the function is called and destroyed automatically when the function is exited.

Storage:	Memory.
Default initial Value:	Garbage.
Scope:	Local (to block in which variable defined)
Lifetime:	Till control remains within block where it is defined.

Example:

```
main()
{
        auto int x, y;
        x = 10;
        printf("Values : %d %d", x, y);
}
```

2. Register Storage Class

We can tell the compiler that a variable should be kept in one of the microprocessor’s register, instead of keeping in memory.

Storage:	CPU Registers.
Default initial Value:	Garbage.
Scope:	Local (to block in which variable defined)
Lifetime:	Till control remains within block where it is defined.

Example:
```
main()
{
       register int a;
       for(a=0;a<10;a++)
            printf(“\nValues : %d ”, a);
}
```

3. Static Storage class

The value of this local variable remains as it is until the end of the program.

Storage:	Memory.
Default initial Value:	Zero.
Scope:	Local (to block in which variable defined)
Lifetime:	Value of the variable remains as it is between function calls.

Example:

```
void insert()
{
       static int m;
       m++;
       printf(“\n%d”, m);
}
main()
{
       insert();
       insert();
       insert();
}
```

4. External Storage class

Accessing the value of the variable which is defined somewhere in the program globally.

Storage:	Memory.
Default initial Value:	Zero.

Scope:	Global
Lifetime:	As long as the program execution doesn't come to an end.

Example:

```
main()
{
   extern int y;
   printf("Value: %d ", y);
}
int y = 30;
```

Flowchart for function

```
main()
{
            int x, y, z;
            printf("Enter values: ");
            scanf("%d%d", &x, &y);
            z = mul(x, y);
            printf("Multiplication: %d", z);
}
int mul(int a, int b)
{
            int m;
            m = a * b;
            return(m);
}
```

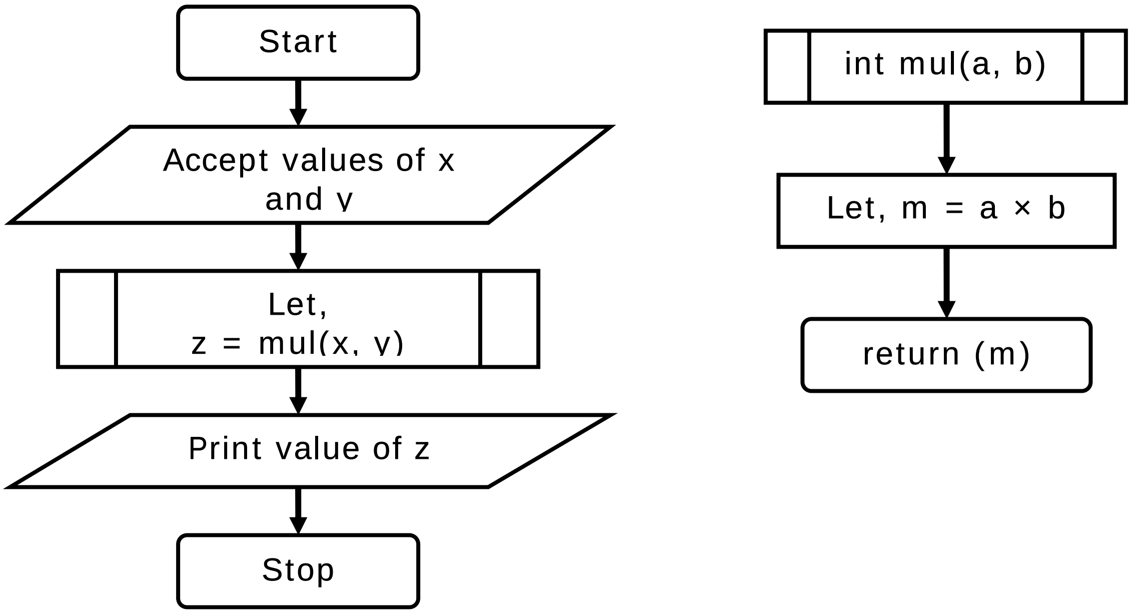


Fig. Flowchart for function

Structure

[Asked in W'07, W'08]

It is a derived data-type in C, which is a collection of different data-type elements. It is a convenient tool for handling a group of logically related data items. Structure helps to organize the complex data in more meaningful way.

Defining a structure

[Asked in W'07, W'08]

Unlike arrays, structures must be defined first for their format that may be used later to declare structure variables. The general format or syntax to create a structure is:

```
struct tag_name
{
     data_type variable1;
     data_type variable1;
     - - - - -;
     - - - - -;
};
```

Remember, this template is terminated by a semicolon. The keyword struct declares a structure to hold the details of different data elements. For example:

```
struct book
{
     char title[20];
     char author[15];
     int pages;
     float price;
};
```

Here, the fields: title, author, pages and price are called as structure elements or members. Each of these members belongs to different data types. The name of structure 'book' is called structure tag. It simply describes a format called template to represent information as shown below:

title	Array of 20 characters
author	Array of 15 characters
pages	Integer
price	Float

Arrays vs Structures

Both arrays and structures are classified as structured data types as they provide mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in number of ways.

1. An array is a collection of related data elements of same type. But structure can have elements of different types.
2. An array is derived data type whereas structure is a user-defined or programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it. But in the case of structure, first we have to design and declare a data structure before variables of that type are declared and used.

Declaring structure variables

[Asked in S'07]

After declaring a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of another type. It includes following elements:

1. The keyword struct
2. The structure tag name
3. List of variable names separated by commas
4. A terminating semicolon

For example, the statement:

```
struct book book1, book2, book3;
```

declares book1, book2, book3 as variables of type struct book. Each of these variables has four members as specified by template. The declaration might look like this:

```
struct book
{
     char title[20];
     char author[15];
     int pages;
     float price;
};
struct book book1, book2, book3;
```

The members of structure themselves are not variables. They do not occupy any memory until they are associated with structure variables such as *book1*. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration,

```
struct book
{
        char title[20];
        char author[15];
        int pages;
        float price;
}book book1, book2, book3;
```

Is valid. Use of tag name is also optional. For example:

```
struct
{
        char title[20];
        char author[15];
        int pages;
        float price;
}book book1, book2, book3;
```

It declares book1, book2, book3 as structure variables representing three books but does not include tag name.

Accessing Structure Variables

Members of structure should be linked to the structure variables in order to make them meaningful members. The link between a member and a variable is established using the member operator '.' which is also known as member operator or period operator. For example:

```
book1.price
```

is the variable representing the price of book1 and can be treated like any other ordinary variable. Here is how we would assign values to the members of book1:

```
strcpy(book1.title, "Programming");
strcpy(book1.author, "T.B.Kute");
book1.pages = 375;
book1.price = 275.00;
```

We can also use scanf to give values through the keyboard.

```
scanf("%s", book1.title);
scanf("%s", book1.author);
scanf("%d", book1.pages);
```

are valid input statements.

Structure Initialization

[Asked in S'07]

Like any other data type, a structure variable can be initialized at compile time.

```
struct time
{
     int hrs;
     int mins;
     int secs;
};
struct time t1 = {4, 52, 29};
struct time t2 = {10, 40, 21};
```

This assigns value 4 to t1.hrs, 52 to t1.mins, 29 to t1.secs and value 10 to t2.hrs, 40 to t2.mins, 21 to t2.secs. There is one-to-one correspondence between the members and their initializing values.

C does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of actual variables.

```
/* Program to declare the structure student having member
variables roll_no, name and age. Accept and display data */
```

```
#include<stdio.h>
main()
{
     struct student
     {
          int roll_no;
          char name[10];
          int age;
     }s;
     printf("Enter roll, name and age: ");
     scanf("%d%s%d", &s.roll_no, s.name, &s.age);
     printf("\nEntered information: \n");
     printf("Roll number: %d", s.roll_no);
     printf("\nName: %s", s.name);
     printf("\nAge: %d",s.age);
}
```

Copying and comparing structure variables

Two variables of the same structure type can be copied the same way as ordinary variables. if person1 and person2 belongs to the same structure, then the following statements are valid:

```
person1 = person2;
```

```
person2 = person1;
```

However, the statements such as,

```
person1 == person2
person2 != person1
```

are not permitted. C does not permit any logical operation on structure variables. In case, we need to compare them, we may do so by comparing members individually.

Operations on individual members

The individual members are identified using the member operator, the dot. A member with dot operator along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators. Consider the following structure,

```
struct person
{
    char name[15];
    int age;
    int salary;
} p1, p2, p3;
```

We can perform following operations on it:

```
if(p1.age > p2.age)
    p1.sarary = p2.salary * 2;
int sum = p1.age + p2.age;
strcpy(p3.name, p2.name);
```

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid:

```
p1.age++;
--p2.salary;
```

Arrays of structure

[Asked in W'07, W'08]

We use structure to describe the format of number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example:

```
struct marks student[60];
```

It defines an array called student that consists of 100 elements. Each element is defined to be of the type *struct marks*. Consider the declaration of the structure marks:

```
struct marks
{
     int subject1;
     int subject2;
     int subject3;
};
main()
{
     struct marks student[3] =
     {{52,96,86}, {42, 49, 52}, {81, 89, 84}};
}
```

This declares the student as an array of three elements student[0], student[1], student[3] and initializes their member as follows:

```
student[0].subject1 = 52;
student[0].subject2 = 96;
student[0].subject3 = 86;
------
------
student[2].subject3 = 84;
```

An array of structures is stored inside the memory in the same way as a multidimensional array. The array will look like as following:

student[0]	student[1]	student[2]	
52	96	86	subject1
42	49	52	subject1
81	89	84	subject1

/* Program to input information of 10 students and display it */

```
#include<stdio.h>
void main( )
{
    struct student
   {
         int roll;
         char name[10];
```

```
          float marks;
    } a[10];
    int i;
    printf(“Enter info. of students:”);
    for(i = 0 ; i < 10 ; i++)
    {
       scanf(“%d %s %f”, &a.roll[i],
              a.name[i], &a.marks[i]);
    }
    for(i = 0 ; i < 10 ; i++)
    {
      printf(“\n%d %s %f”,
              a.roll[i], a.name[i], a.marks[i]);
    }
 }
```

Review Questions

Summer 2007: **Marks = 12**

a) Define function prototype. 2

b) Declare and define a structure employee having member variables as emp_id, emp_name & salary. 2

c) Explain with example: structure initialization and declaration. 4

d) Write a program to declare the structure student having member variables roll_no and name. Accept data for three students and display it. 4

Winter 2007: **Marks = 12**

a) Give the syntax of declaring structure. 2

b) Define function. 2

c) Explain what is structure and array of structure. Declare a structure with elements as roll_no and name. Declare array of structure for 10 students. 4

d) Declare a structure ‘book’ having data members as title, author and price. Accept this data for one structure variable and display accepted data. Write a program. 4

Summer 2008: **Marks = 16**

a) Explain function definition. Give syntax for function. 2

b) Give general form of structure and define a structure student having member variables as roll_no, name and dob. 2

c) Write the output of following program. Take suitable input. 4

```
#include<stdio.h>
void large( )
{
     int a,b;
     printf(“Enter values of a and b :”);
     scanf(“%d %d”,&a,&b);
     if(a<b)
     printf(“large : %d”,b);
     else
     printf(large : %d”,a);
}
main( )
{
     large( );
     large( );
}
```

d) Define two structures date and account.
Date has members: day, month, year
Account has members as: acc_no, balance, dob
Assign initial values to them. 4

e) Declare a structure ‘Book’ having data members title, author and price. Accept this data for one variable and display accepted result. 4

Winter 2008: **Marks=16**

- **a)** What do you by structure? Give syntax of declaring it. 2
- **b)** Define recursion. 2
- **c)** What is function? Explain the need of function. 4
- **d)** Explain with example the array of structure. 4
- **e)** Write a program to define a structure employee with members emp_name,
  emp_id and salary. Accept data for one employee and display it. 4

Other: **Marks**

- **a)** Explain recursion with example 4
- **b)** State types of the functions. 2
- **c)** Describe the use of static and extern variables. 4
- **d)** Explain the meaning of register and auto storage classes. 4
- **e)** Define scope, lifetime & visibility of variables. 2
- **f)** Differentiate structure and array with example. 4
- **g)** Give syntax of declaring a function. 2
- **h)** Define global and local variables. 2

Programming Exercises

1. Create a function add( ) which will calculate the sum of the array elements passed as parameters to it.

2. Write a function to calculate the factorial of the number.

3. Create a user defined function to find the minimum value of three float numbers which are passed as the arguments to it.

4. Write a function range( ) which will take value of A and B as parameter and displays total numbers divisible by 3 in the range A to B.

5. Write a function which will take a year as argument and find whether that year is a leap year or not?

6. Create a function to pass any string as argument. This function will find the total number of vowels in that string. Take integer as the return type of the function.

7. Define a structure Cricket having data members as team name, player name and average. Accept the values of 5 different players and display their data team-wise.

8. Define a structure Nation contains data members as name, capital and population. Accept the data for 10 nations and display it in reverse order.

9. Define a structure Book having data members as book name, author, pages and price. Accept this data for 7 different books and display those data having pages greater than 300.

10. Declare two structures named Date and Time. Date contains day, month and year whereas Time contains hours, minutes and seconds. Accept the date and time and display date in *month day, year* (e.g. February 9, 2009) and time in *hh:mm:ss* (e.g. 12:56:02) format.

11. Define a structure Student having data members as name, age, class and marks. Accept this data for 5 students and display this data for student who is youngest?

12. Define a structure College having data members as name, branches, result. Accept this data for 5 colleges and identify whose result is worst?

References

1. **Programming in ANSI C**
   by, E Balagurusamy, Tata McGraw Hill, 4th Edition
   Chapter 9: User-defined functions
   Chapter 10: Structures and Unions

Chapter 05

Pointers

Lectures allotted: 14

Marks Given: 16

Contents:

Pointers

[Asked in S'07, W'07, S'08, W'08]

A pointer is a derived data type in 'C'. It is built from any one of the primary data type available in 'C' programming language. Pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where the program instructions and data are stored. Pointers can be used to access and manipulate data in memory.

Advantages of Pointers

1) Pointers are most efficient in handling arrays.
2) They can be used to return multiple values from function through function argument.
3) They permit references to functions and thereby they provide facility to pass function as argument to another function.
4) The use of pointer arrays to character string saves the data storage space in the memory.
5) Pointers allow C to supports dynamic memory management.
6) They provide an efficient tool for manipulating dynamic data structures such as structures, link lists, quos, stacks, trees and graphs.
7) Pointers reduce the complexity and length of the program.
8) Pointers increase the execution speed and thus reduce the program execution time. Of course the real power of C lies in the proper use of pointers.

Understanding Pointers

Computer memory is sequential collection of storage cells as shown in figure. Each cell is commonly known as byte and has a number called as address associated with it. Typically the addresses are numbered sequentially starting from zero. The last address depends upon the memory size. A computer having 64KB memory will have its last address as 65535.

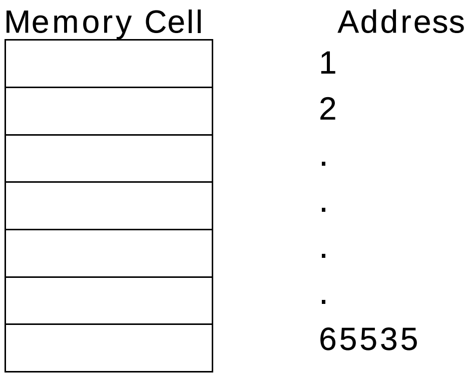
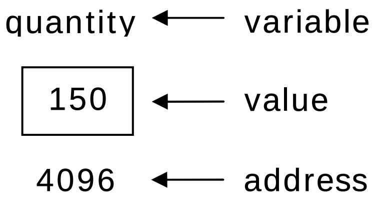


Fig. Memory Representation

Whenever we declare variable, the system allocates it somewhere in the memory. An appropriate location holds the value of variable. Since every byte has a unique address number, this location will have its own address number. Consider the following statement:

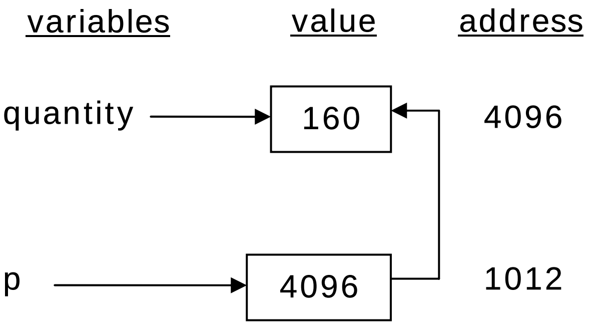
```
int quantity=150;
```

This statement instructs the system complier system to find a location for integer variable quantity and store value 150 in that location. Let us assume that the system has chosen the address location 4096 for *quantity*, so it will be visualized as:

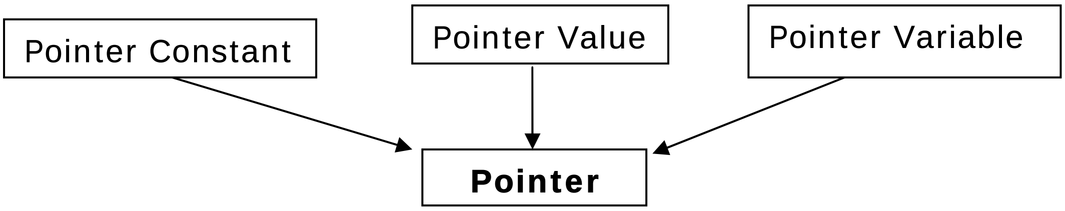


During the execution of program computer always associates the name quantity with address 4096. We may have access to the value 150 by using either the name *quantity* or the address 4096. Since memory addresses are simply numbers they can be stored in memory like any other variables. Such variables that hold memory addresses are called as 'pointer variables' the pointer variable is nothing but a variable that contains an address which is location of another variable in memory.

Remember: pointer is a variable. Its value is stored in memory in another location. Suppose we assign the address of quantity to a variable 'p', the link between the variable 'p' and quantity can be visualized as:



Since the value of variable 'p' is the address of variable *quantity* for using the value of 'p' and therefore we say that the variable p points to the variable *quantity*. Thus p is called as 'pointer'.



The memory addresses within a computer are referred as pointer constants. We cannot change them; we can only use them to store the data values. The value that is stored as the memory location pointed by pointer is known as pointer value. It can be changed. The pointer value can be stored in another variable. The variable that contains the pointer value is called as pointer variable.

Accessing the address of variable

The actual location of variable in the memory is computer dependent and therefore the address of variable is unknown to immediately. If we want to identify address any variable we can use the unary operator & of C programming language. It is known as 'address of' operator. [Asked in W'07]
Example:

```
int x=10;
printf("%u",&x);
```

These statements will print the address of variable x where it is stored in the memory. The & operator can only be used with a simple variable or an array element. Following are the illegal use of & operator

```
int x[10];
&9;      /* invalid statement */
&x;      /* invalid statement */
&(x+y);  /* invalid statement */
```

Declaring pointer variable

[Asked in W'07, S'08, W'08]

In C every variable must be declared for its data type. Since pointer variables contain addresses that belong to a separate data type they must be declared as pointers. The declaration of pointer variable takes following form:

```
data-type *variable-name;
```

The statement tells that the *variable-name* is a pointer variable of *data-type* which will store address of the same *data-type* variable. For example:

```
int *p;
```

It declares the variable 'p' as a pointer that will point to an integer data type variable. The p can store address of another integer variable only. Similarly,

```
float *x;
```

It declares x as pointer variable a floating point type.

Initialization of pointer variable

[Asked in S'07, W'07]

The process of assigning the address of a variable to a pointer variable is known as initialization. For example:

```
int *p;
int x = 10;
p = &x;
```

Pointer declaration styles

Pointer variables are declared similar to normal variable except for addition of unary * operator. The symbol can appear anywhere between the data type name and the pointer variable name such as,

```
int * p;      /*style 1 */
int *p;       /*style 2 */
int *    p;   /*style 3 */
```

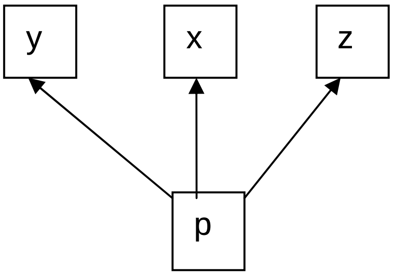
Multiple declarations

```
int *p, x, *q;
```

Pointer's flexibility

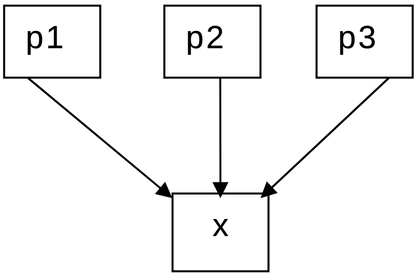
Pointers are flexible. We can make the same pointer to point different data variables in different statements.
Example:

```
int x, y, z, *p;
. . . . . . .
p = &x;
. . . . . . .
p = &y;
. . . . . . .
p = &z;
```



We can also use different pointers to point the same data variable. For example:

```
int x;
int *p1=&x;
int *p2=&x;
int *p3=&x;
. . . . . .
. . . . . .
```



Assessing a variable through pointer

Once a pointer has been assigned address of a variable, the value of that variable can be accessed using the pointer. This is done by using unary operator * (asterisk) usually known as indirection operator, dereferencing operator or 'contain of' operator. For example:

```
int marks,*p,n;
marks=79;
p=&marks;
n=*p;
```

The first line declares *marks* and *n* as integer variables and *p* as a pointer variable pointing to an integer. The second line assigns value 79 to variable *marks* and third line assigns address of variable *marks* to pointer *p*. The fourth line contains indirection operator *. When * is placed before a pointer variable in an expression, the pointer returns value of the variable of pointer. In above case *p returns value of variable *marks* because *p* contains the address of *marks*. The * can be remembered as 'value at address'. Thus value of *n* would be 79.

```
p=&marks;
n=*p;
```

is equivalent to:

```
n=*&marks;
```

Chain of pointers

It is possible to make a pointer to point to another pointer thus creating a chain of pointer as shown below:

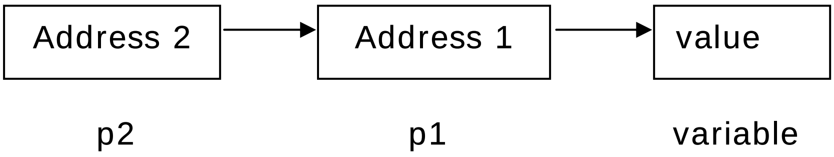


Fig. Multiple Indirections

Here a variable *p2* contains the address of pointer variable *p1* which points to the location that contains the desired value. This is known as multiple indirections. A variable that is a pointer to a pointer must be declared using additional indirection operation in front of the name. Example:

```
int **p;
```

This declaration tells us that *p2* is a pointer to a pointer of int type. Remember, the pointer *p2* is not a pointer to an integer but a pointer to a pointer integer. We can access the variable value indirectly pointed by pointer by applying the indirection operation twice in front of the double pointer. Example:

```
void main()
{
     int x,*p,**p;
     x=100;
     p1=&x;          /* pointer to variable*/
     p2=&p1;         /* pointers to pointer*/
     printf(“%d”,**p2);
}
```

Output: 100

Pointer’s Arithmetic

[Asked in S’07, W’07, S’08, W’08]

Addition on pointers

1. Like other variables pointer variables can be used in arithmetic operations. Example:

   ```
   int *p, x=10;
   p = &x;
   *p=*p+5;
   ```

   This statement with add the value 5 in the value pointed by p.
2. Integer pointers are incremented or decremented in the multiples of 2. Similarly character by 1, float by 4 and long pointers by 8 etc.

   ```
   p++;                /* valid */
   ```

3. We cannot add or subtract one pointer from another. Example:

   ```
   int *p1, *p2;
   p1 = p1 + p2;       /* invalid */
   ```

4. We can add or subtract the constant from pointer variable. In this case value of pointer is incremented or decremented by the scaling factor of respective data type. Example:

   ```
   p = p + 2;          /* valid */
   ```

Multiplication and Division on pointers

1. The value stored at pointers Address can be added or subtracted by any constant value. Example:

   ```
   *p = *p * 4;  /* valid */
   ```

2. We cannot use two pointers for multiplication or division. That is,

   ```
       x = x * y;
       x = x / y;
   ```

   is not allowed.
3. We cannot multiply or divide a pointer by constant. Example:

   ```
       p = p * 4;     /* invalid */
       p = p / 2;     /* invalid */
   ```

Rules for pointer operation

1. A pointer variable can be assigned address of another variable.
2. Pointer variable can be assigned value of another pointer variable.
3. Pointer variable can be initialized by NULL value of zero value.
4. Pointer variable can be used with pre increment or decrement or post increment or decrement operators.
5. An integer value may be subtracted or added from the pointer variable.
6. When two pointers point to the same array one pointer variable can be subtracted from another.
7. When two pointers point to the object of same data type we can use the relational operators to compare them.
8. Pointer variable cannot be multiplied or divided by a constant.
9. Two pointer variables cannot be added, subtracted, multiplied or divided.
10. A value cannot be assigned to any particular address. Example:

    ```
    int a=20;
    &a=1000;   // It is illegal.
    ```

Pointers and arrays

[Asked in S'07, W'07, S'08]

When an array is declared the compiler allocates base address and sufficient amount of storage memory to contain all the elements of the array in contiguous memory location. The base address is the location first element that is 0th element of the array. The compiler also defines the array name as a constant pointer to the first element. For example:

```
    int x[5] = {8, 4, 9, 6, 3};
```

Suppose the base address of x is 9092 and assuming each integer requires 2 bytes the five elements are stored as shown below . . . .

Index	Value	Address
x[0]	8	9092
x[1]	4	9094
x[2]	9	9096
x[3]	6	9098
x[4]	3	9100

The name x is defined as a constant pointer pointing to the first element x[0] and therefore value of x is 9092. If we declare p as an integer pointer then we can make the pointer p to point the array x by following assignment.

```
        int *p;
        p = &x[0];
        OR
        p = x;
```

Now we can access every value of x using p++ to move from first element to another i. e.

```
        p = &x[0];    =    9092
        p = &x[1];    =    9094
        p = &x[2];    =    9096
        p = &x[3];    =    9098
        p = &x[4];    =    9100
```

For initializing an array at run time:

```
    for(i=0;i<5;i++)
       printf("\n%d",*(p+i));
```

```
/* Program to find the average of five students using pointers */

        #include<stdio.h>
        main()
        {
            int x[5],*p, sum = 0,i;
            for(i=0;i<5;i++)
               scanf("%d", &x[i]);
            p=x;
            for(i=0;i<5;i++)
               sum=sum + *(p+i);
```

```
                printf(“%d”, sum/5);
        }
```

Pointers to String

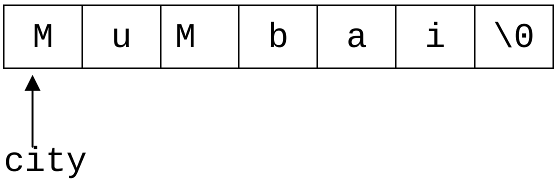
Strings are created like character array. Therefore, they are declared and initialized as:

```
char city[10] = “Mumbai”;
```

M	u	M	b	a	i	\0	\0	\0	\0

Compiler automatically inserts NULL character ‘\0’ at the end of a string. ‘C’ supports an alternative method to create strings using pointer variables of type ‘char’. Such as,

```
char *city = “Mumbai”;
```



M	u	M	b	a	i	\0

This creates a string and then stores its address in the pointer variable *city*. This pointer variable now points to the first character of the string. That is, the statements:

```
printf(“%c”,*city);
```

will print the current pointing character ‘M’ on the screen and the statement,

```
printf(“%s”, city);
```

will print all the sets of characters from current pointing location to the first occurrence of ‘\0’. We can also use the runtime assignment for initializing the pointer such as,

```
char *city;
city = “Mumbai”;
```

This is not a string copy. It will only copy the address of first constant from array i.e. ‘M’ into the character pointer *city*. We can also use scanf statement to initialize the string using pointers. Example:

```
     char *name;
     printf(“Enter your name: ”);
     scanf(“%s”, name);
     printf(“%s”, name);
          OR
     while(*name!= ‘\0’)
     {
          printf(“%c”,*name);
          name++;
     }
```

/* Program to input the string and print it in reverse order */
[Asked in W’07]

```
     char *str;
     int len = 0;
     printf(“Enter the String: ”);
     gets(str);
     for( ;str!=’\0’;str++);
        str--;
     for( ;len>0;len--,str--)
        printf(“%c”,*str);
```

Array of Pointers

[Asked in W’08]

We can create the array of pointers also with the same declaration of the array. For example:

```
     int *a[10];
```

here, the array of pointers will be created i.e. the array will point to 10 different integers, such as:

```
     a[0]=&x;
     a[1]=&y;
     a[2]=&z;
          .
          .
          .
```

Example:

```
     for(i=0;i<10;i++)
          scanf(“%d”,a[i]);
     for(i=0;i<10;i++)
          printf(“%d”,*a[i]);

     char nations[3][3];
```

This declaration says that 'nations' is a table containing 3 names each with maximum length of 15 characters. So, the total storage requirement for this array is 45 bytes. We know that very rarely the individual string will be of equal lengths. Therefore instead of making each row of a fixed number of characters we can make it a pointer to a string of variable length.Example:

```
char *nations[3]={"India", "Australia", "New Zealand"};
```

This declares nations to be an array of three pointers each pointing to particular names as:

```
name[0]       "India"
name[1]       "Australia"
name[2]       "Newzeland"
```

This declares declaration allocates only 25 bytes of memory to store the string. Following statement will print the strings on the screen:

```
for(i=0;i<3;i++)
    printf("\n %s", nations[i]);
```

Pointers to Function

When an array is passed as function argument only the address of first element of the array is passed but not the actual value of array elements. When we pass address to a function the parameters receiving the address should be pointers. The process of calling a function using pointers is called as 'call by reference'. In general when we call a function is called as call by value. [Asked in S'08] Example:

```
change(int x)
{
     x=x+10;
}
main()
{
     int a=5;
     change(a);    //call by value
     printf("%d", a);
}
```

Here function change() is called by value in the function main(). The 'a' is the actual parameter of the function and 'x' is the formal parameter of the function. When the function is called by value, the value of variable 'a' is passed into variable 'x'. Therefore, it is called as 'call by value'. The 'x' & 'a' are the local variables of individual functions that is main() and change(). Therefore, the changes made on variable 'x' will not affect variable 'a'. Therefore the 'call by value' doesn't reflect the change. Instead of doing this we call the function by reference that is by passing addresses of the variable. The Function; which is called by reference can change value of variable used in the call. Example:

```
change(int *x)
{
     *x=*x+10;
}
main()
{
     int a+5;
     change(&a);  // call by reference
     printf("%d",a);
}
```

When the function change() is called, the address of variable 'a' (not the value) is passed into the function change(). Inside change(), variable 'x' is declared as a pointer. Therefore 'x' will contain the address of variable 'a'. The statement:

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

means add 10 to the value stored at address of 'a' that is 'x'. Since 'x' represents the address of 'a', value of 'a' is changed from 5 to 15. Therefore call by reference provides mechanism by which the function can change the stored value in the calling function. This mechanism is also known as 'call by address' or 'pass by pointers'. Example:

[Asked in S'08 twice, W'08]

```
void swap(int *x, int *y)
{
     int t;
     t = *x;
     *x = *y;
     *y = t;
}
main()
{
     int a = 5, b = 10;
     swap(&a, &b);
     printf("%d%d", a, b);
}
```

Example:

```
void copy(int *n, int *m)
{
     int i;
     for(i=0;i<10;i++, n++, m++)
```

```
          *n = *m;
    }
    main()
    {
       int a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
       int b[10];
       copy(b, a);       //call by reference
       for(i=0;i<10;i++)
         printf("\n%d", b[i]);
    }
```

Differences between Call by Value and Call by Reference

[Asked in S'07, W'07]

Sr. No.	Call by Value	Call by Reference
1	The function is called by directly passing value of variable as argument	The function is called by directly passing address of variable as argument
2	We need to declare a general variable as function argument	We need to declare a pointer variable as argument.
3	Calling function by value does not changes actual values of variables	Calling function by reference changes actual values of variables
4	It is a slow way of calling function as we are calling it by passing value	It is a fast way of calling function as we are calling it by passing address of a variable

5	For example:
	Change(int p)
	{
	p = p + 10
	}
	Call:
	Change(a);
		For example:
		Change(int *p)
		{
		*p = *p + 10
		}
		Call:
		Change(&a);

Review Questions

Summer 2007: **Marks = 28**

a) Justify following statements: 2

```
int a = 10;
int *b = &a;
int *c = b;
int d = *b + *c;
```

b) What is pointer? 2

c) Distinguish between call by value and call by reference (W'07). 4

d) Write a program using pointers to compute sum of all elements stored in an array. 4

e) What will be the output of the following program? 4

```
#include<stdio.h>
void main ( )
{
   float a = 13.5;
   float *b, *c;
   b = &a; c = b /* suppose address of a is 1006*/
   printf("%u %u %f %f", &a, b, *(&a), *b);
}
```

f) Explain use of pointers and arrays. 4

g) Explain the concept of pointer's arithmetic operations with examples. (W'07) 4

h) How pointer is initialized? 4

Winter 2007: **Marks = 28**

a) What is pointer variable? 2

b) Give syntax of declaring and initializing pointer. 2

c) Explain concept of array with pointer. 4

d) Write a program to reverse the string using pointer. 4

e) Explain the difference between call by value and call by reference methods for calling function. 4

f) Explain the concept of pointer arithmetic operations. 4

g) What is pointer? Give the use of ampersand (&) and asterisk (*) operators in pointer. 4

h) Explain the meaning of following statement with reference to pointers. 4

```
int *P, X;
X = 10;
*P = X;
P = &X;
```

Summer 2008: **Marks = 28**

a) Define call by value. 2

b) What is pointer? Give its declaration. 2

c) Write a simple program for accessing the data through pointer variable. 4

d) Two numbers are input through keyboard into two locations C and D. Write a program to interchange the contents of C and D. 4

e) Write a program for exchanging values of two variables using call by reference. 4

f) Explain concept of pointer arithmetic operations. 4

g) Explain how array elements are accessed by using pointers. 4

h) What will be the output of the following program? 4

```
#include<stdio.h>
void main( )
{
        int i = 3; int * j;
        j = &i;
        printf("\n Address of i = %u", &i);
        printf("\n Address of i = %u", j);
        printf("\n Address of j = %u", &j);
        printf("\n value of j = %u", j);
        printf("\n value of i = %d", i);
        printf("\n value of i = %d", *(&i));
        printf("\n value of i = %d", *j);
}
```

Winter 2008: **Marks = 24**

a) Give the meaning of declaration int *ptr; 2

b) What is pointer? 2

c) Write a program using pointer to accept an integer array and print it in reverse order. 4

d) Explain the concept array of pointers with example. 4

e) Write a function to swap (exchange) the values of two integer numbers using pointers. 4

f) Explain pointer arithmetic with example. 4

g) Explain the meaning of following statements with reference to pointer:

```
int *ptr, m = 8;
*ptr = m;
ptr = &m;
```

4

Programming Exercises

a) Write a program to reverse the string using pointers.

b) Find the largest number from a float array using pointers.

c) Calculate the addition of two 4 X 4 matrices using pointers. Take the input from user.

d) Print the Fibonacci series using pointers.

e) Compare two string using pointers and concatenate them if both of them are equal.

f) Sort the character array in descending order using pointers.

g) Exchange the values of two variables using pointers.

h) Count the number of vowels in a string using pointers.

i) Print the array of integers using pointers to the array.

j) Calculate the addition of all the numbers stored in an array using pointers.

k) Find the number of occurrences of specific character in the string. (Make the use of pointers).

l) Declare the array of 20 integer numbers. Find and display the average of only even numbers by using pointers.

m) Check whether the string is palindrome or not, make the use of pointers.

n) Inverse the case of characters of a string by using pointers. Upper case letters must be converted to lower case and lower case to the upper case.

o) Analyze the string i.e. take the string as the input from user and count number of characters, numbers and special symbols in it. (Hint: Make the use of ASCII equivalent of the characters.)

p) Take the sentence as the input and count number of words existed in it.

q) Sort the array of strings by using pointers.

r) Print the ASCII equivalents of the characters from the string by using pointers.

s) Check whether the string inputted is a valid variable name or not. That is, check whether it is following all the rules of creating variables name or not?

t) Print the addresses of all the elements stored in the integer array, float array and character array.

u) Accept the string as the input from user and replace a specific character with another specific character. Take all three inputs from user.

v) Input an arithmetic expression from keyboard and check whether it contains number of opening brackets equal to closing brackets or not.

w) Find all the prime numbers from 1 to 100 using pointers.

x) Pass the whole array to a function and print it using pointers.

y) Input a string from keyboard and check whether it is starting and ending with a number or not?

z) Compare two strings are equal or not by ignoring the case.

References

1. **Programming in ANSI C**
   by, E Balagurusamy, Tata McGraw Hill, 4th Edition
   (All the data is referred from this book, Chapter No. 11 Pointers)

   You can use the following dedicated book on pointers also:

   Understanding Pointers in C
   by, Yashvant Kanetkar, BPB Publications, First Edition, 2006

Notes

- Most of the books used as reference for the chapters are available in e-copy on the following sites. You can use them for reference too.
  - www.esnips.com
  - www.4shared.com
  - www.isbnonline.com
  - www.pdfchm.com
  - www.scribd.com
- The reference material that is not been included in this book such as transparencies, objective questions, notes, MSBTE and Sample question papers, program's soft copy are available on following link of Google sites:

  http://sites.google.com/site/tusharkute
- The following software are used for compiling programs:

  Borland Turbo C++ Compiler
- The following software are used for formatting this documents:

  Editing: Microsoft Office Word 2003, Windows notepad.

  PDF Creation: Adobe PDF Maker 6.0
- By analyzing last 4 questions papers of MSBTE, I have analyzed the format of the paper. This will be beneficial for the students learning diploma in Computer group under MSBTE. It is given on the next page.
- You are free to send the queries and suggestions at:

  tbkute@gmail.com

Programming in 'C' (9017)
Question Paper Analysis

Overall Questionnaire:

Chapter	**Marks**
Chapter 1	16 / 20
Chapter 2	32 / 36
Chapter 3	16
Chapter 4	12 / 16
Chapter 5	28
Total	**108**

Programs:

	Ch: 1	Ch: 2	Ch: 3	Ch: 4	Ch: 5	Total
Summer 07	--	16	04	04	04	**28**
Winter 07	--	16	04	04	04	**28**
Summer 08	--	16	04	04	08	**32**
Winter 08	04	12	04	04	08	**32**
Summer 09	**?**	**?**	**?**	**?**	**?**	**?**

Compulsory Questions

1. Operators

   02 + **04** Marks

2. If-else, else-if

   04 Marks

3. Difference between while and do-while

   02 / 04 Marks

4. The switch statement

   02 / 04 Marks

5. break / continue

   02 Marks

6. 2D Array

   04 Marks

7. Two string library functions

   02 + 04 Marks

8. A Program on simple structure

   04 Marks

9. Call by reference

   04 Marks

10. Pointer Arithmetic

    04 Marks

11. What is pointer?

    02 Marks

All the Best
