PROGRAMMING IN C

INTRODUCTION TO C PROGRAMMING

THE C CHARACTER SET

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces

The complete character set is given below

Letters

- Uppercase A - Z
- Lowercase a – z

Digits

- All decimal digits 0 – 9

Special characters

, comma	# number sign
. period	& ampersand
; semicolon	^ caret
: colon	* asterisk
? question mark	- minus sign
‘ apostrophe	+ plus sign
“ quotation mark	< opening angle bracket or less than sign
! exclamation mark	> closing angle bracket or greater than sign
| vertical bar	
/ slash	(left paranthesis
\ backslash	) right paranthesis
~ tilde	[left bracket
_ under score	] right bracket
$ dollar sign	{ left brace
% percent sign	} right brace

White spaces

- Blank space
- Horizontal tab
- Carriage return
- New line
- Form feed

Trigraph characters

ANSI C introduces the concept of “trigraph” sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in the following table.

Trigraph sequence	Translation
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vertical bar
??/	\ back slash
??’	^ caret
??-	~ tilde

IDENTIFIERS AND KEYWORDS

Identifiers are names that are given to various program elements, such as variables, functions and arrays. The rules for identifiers are

1. Both uppercase and lowercase letters are permitted.
2. Digits 0 to 9.
3. They must begin with a letter.
4. Uppercase and lowercase letters are significant.
5. The underscore character (_) can also be included, and is considered to be a letter. An identifier may also begin with an underscore.
6. The variable name should not be a keyword.
7. White space is not allowed.

The following names are valid identifiers

X	y12
sum_1	_temp
names	area
tax_rate	TABLE

The following names are not valid identifiers for the reasons stated.

4th	The first character must be a letter.
“x”	Illegal character(“)
order-no	Illegal character(-)
error flag	Illegal character (blank space)

There are certain reserved words, called keywords, that have standard, predefined meanings in C. These keywords can be used only for their intended purpose; they cannot be used as programmer defined identifiers.

The standard keywords are

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

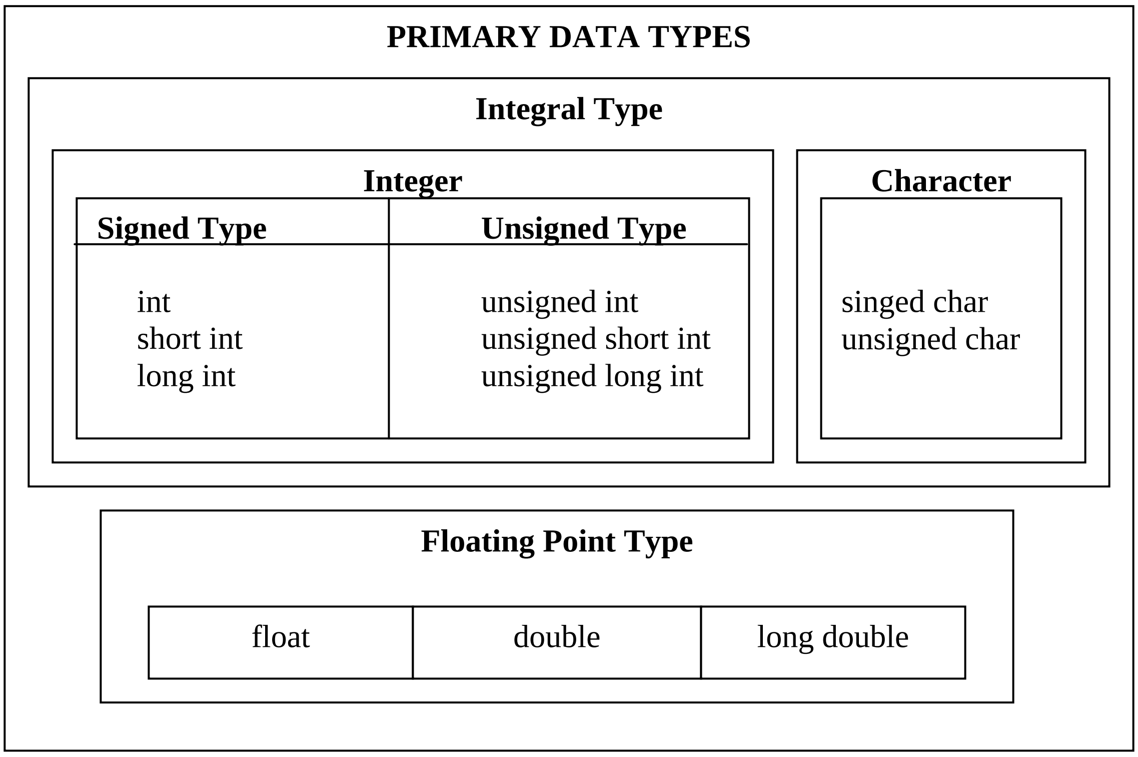
DATATYPES

C supports several different types of data, each of which may be represented differently within the computer's memory. ABSU C supports four classes of data types:

1. Primary (or fundamental) data types
2. User-defined data types
3. Derived data types
4. Empty data set

All C compilers support four fundamental data types,

- integer (int)
- character (char)
- floating point (float)
- double-precision floating point (double)



The range of the basic four types are given below

Data Type	Range of values
char	-128 to 127
int	-32,768 to 32,767
float	3.4e-38 to 3.4e+38
double	1.7e-308 to 1.7e+308

Integer Types

Integers are whole numbers with a range of values supported by a particular machine. C has three classes of integer storage, namely short int, int, and long int, in both signed and unsigned forms. The following table shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

Type	Size (bits)	Range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int or signed int	16	-32,768 to 32,767
unsinged int 16	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E-38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 1.1E+4932

Floating Point Types

Floating point (or real) numbers are stored in 32 bits, with 6 digits of precision. Floating point numbers are defined in C by the keyword float. When the accuracy provided by a float number is not sufficient, the type double can be used to define the number. A double data type number uses 64 bits giving a precision of 14 digits.

Character Types

A single character can be defined as a character (char) type data. Characters are usually stored in 8 bits of internal storage. The qualifier signed or unsigned may be explicitly applied to char. While unsigned chars have values between 0 to 255, signed chars have values from -128 to 127.

CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. There are four basic types of constants in C. They are

1. Integer constants
2. Floating-point constants
3. Character constants and string constants
4. Enumeration constants

Integer and floating-point constants represent numbers. They are often referred to collectively as numeric-type constants. The following rules apply to all numeric-type constants.

1. Commas and blank spaces cannot be included within the constant.
2. The constant can be preceded by a minus (-) sign.
3. The value of a constant cannot exceed specified minimum and maximum bounds.

Integer Constants

An integer constant is an integer-valued number. Thus, it consists of a sequence of digits. Integer constants can be written in three different number systems:

1. decimal (base 10)
2. octal (base 8)
3. hexadecimal (base 16)

A **decimal integer constant** can consist of any combination of digits taken from the set 0 through 9. If the constant contains two or more digits, the first digit must be something other than 0.

Several valid decimal integer constants are shown below

```
0    1    23    5423    -234
```

The following decimal integer constants are not valid for the reasons stated.

12,123	Illegal character (,)
23.0	Illegal character (.)
23 34 45	Illegal character (blank space)
23-34-344	Illegal character (-)
0934	The first digit cannot be a zero.

An **octal integer constant** can consist of any combination of digits taken from the set 0 through 7. However the first digit must be 0, in order to identify the constant as an octal number.
Several valid octal integer constants are shown below

0	01	023	05423	0777

The following octal integer constants are not valid for the reasons stated.

234	Does not begin with 0
02390	Illegal digits (9)
777.77	Illegal character (.)

A **hexadecimal integer constant** must begin with either 0x or 0X. It can then be followed by any combination of digits taken from the sets 0 through 9 and a through f (or A through F).
Several valid hexadecimal integer constants are shown below

0x	0X1	0X23F	0xFF	0xabcd

The following hexadecimal integer constants are not valid for the reasons stated.

0x 12.34	Illegal character (.)
0BE38	Does not begin with 0x or 0X
0x.4bff	Illegal character (.)
0XDEFG	Illegal character (G).

Unsigned and Long Integer Constants

Unsigned integer constants may exceed the magnitude of ordinary integer constants by approximately a factor of 2, through they may not be negative. An unsigned integer constant can be identified by appending the letter U(or u) to the end of the constant.

Long integer constants may exceed the magnitude of ordinary integer constants, but require more memory with the computer. A long integer constant can be identified by appending the letter L (or l) to the end of the constant.

An unsigned long integer may be specified by appending the letters UL (or ul) to the end of the constant.

Several unsigned and long integer constants are sown below.

Constant	**Number System**
50000U	Decimal (unsigned)
123456789L	Decimal (long)
123456789UL	Decimal (unsigned long)
0123456L	Octal (long)
0777777U	Octal (unsigned)
0x50000U	Hexadecimal (unsigned)
0XFFFFFUL	Hexadecimal (unsigned long)

Floating-point constants

A floating-pint constant is a base-10 number that contains either a decimal point or an exponent (or both).

Several valid floating-point constants are shown below.

0.	1.	0.2	827.324	5000.
0.00073	2E-7	0.006e-3	1.77e+7	.1212e5

The following are not valid floating-point constants for the reasons stated.

1	Either a decimal point or an exponent must be present.
1,000.0	Illegal character (,)
2E+7.2	The exponent must be an integer quantity
3E 10	Illegal character (blank space) in the exponent.

If an exponent is present, its effect is to shift the location of the decimal point to the right, if the exponent is positive, or to the left, if the exponent is negative. Example:

The quantity $3X10^5$ can be represented in C by any of the following floating-point constants.

300000.	3e5	3e+5	3E5	3.0e+5
.3e6	0.3E6	30E4	30.E+4	300e3

Floating-point constants are normally represented as double-precision quantities in C. Hence each floating-point constant will typically occupy 2 words (8 bytes) of memory. Some versions of C permit the specification of a “single-precision” floating-point constant, by appending the letter F (or f) to the end of the constant (eg. 3.5F). Similarly, some versions of C permit the specification of “long” floating-point constant, by appending the letter L (or l) to the end of the constant (e.g. 0.123456789E-33L).

Character Constants

A character constant is a single character, enclosed in apostrophes (‘).
Several character constants are shown below.

‘A’ ‘x’ ‘3’ ‘?’ ‘ ‘

Character constants have integer values that are determined by the computer’s particular character set. Thus, the value of a character constant may vary from one computer to another.

Several character constants and their corresponding values, as defined by the ASCII character set, are shown below.

Constant	**Value**
‘A’	65
‘x’	120
‘3’	51
‘?’	63
‘ ‘	32

Escape Sequences

Certain nonprinting characters can be expressed in terms of escape sequences. An escape sequence always begins with a backward slash and is followed by one or more special characters.
The following are the commonly used escape sequences.

Escape Sequence	**Character**
\a	Bell (alert)
\b	backspace
\t	horizontal tab
\v	vertical tab
\n	newline (line feed)
\f	form feed
\r	carriage return
\”	quotation mark (“)
\’	apostrophe (‘)
\\	backslash (\)
\0	null

String Constants

A string constant consists of any number of consecutive characters enclosed in double quotation marks.

Several string constants are shown below.

“green”	“A”	“123-23-345”
“Rs.123.50”	“SRM University”	“Welcome to C programming”

The compiler automatically places a null character (\0) at the end of every string constant. This character is not visible when the string is displayed.

VARIABLES AND ARRAYS

A variable is an identifier that is used to represent a single data item. The data item must be assigned to the variable at some point in the program. The data item can then be accessed later in the program simply by referring to the variable name.

The array is an identifier that refers to a collection of data items that all have the same name. The data items must all be of the same type. The individual data items are represented by their corresponding array elements. The individual array elements are distinguished from one another by the value that is assigned to a subscript.

Example:
Suppose that x is a 10-element array. The first element is referred to as x[0], the second as x[1], and so on. The last element will be x[9].

There are different ways to categorize arrays, integer arrays, character arrays, one-dimensional arrays, multi-dimensional array. Since the array is one-dimensional, there will be a single subscript (sometimes called an index) whose value refers to individual array elements. If the array contains n elements, the subscript will be an integer quantity whose values range from 0 to n-1. Note that an n-character string will require an (n+1) element array, because of the null character (\0) that is automatically placed at the end of the string.

DECLARATIONS

A declaration associates a group of variables with a specific data type. All variables must be declared before using the variable in the program. The syntax of declaration statement is

Data-type variable1,variable2,…,variableN;

A declaration consists of a data type, followed by one or more variable names, ending with a semicolon.

Example:

```
int a,b,c;
float pi,area,avg;
char c;
short int m1,m2,m3;
long fact;
unsigned x,y;
double root1,root2;
```

Initial values can be assigned to variables within a type declaration. To do so, the declaration must consist of a data type, followed by a variable name, an equal sign (=) and a constant of the appropriate type.

Example:

```
int a=10,b=20;
char c=’*’;
float sum=0.0;
double factor=0,21023e-6;
```

A character-type array can also be initialized within a declaration.

Example:

```
char text[]=”India”;
```

This declaration will cause text to be an 6-element character array. The first 5 elements will represent the 5 characters within the word India, and the 6th element will represent the null character (\0).

The declaration could also have been written

```
char text[6]=”India”;
```

where the size of the array is explicitly specified.

EXPRESSIONS

An expression is a combination of variables, constants, and operators arranged as per the syntax of the language. Expressions can also represent logical conditions that are either true or false.

Example:

```
a + b
x = y
c = a + b
x <= y
x == y
++i
```

STATEMENTS

A statement causes the computer to carry out some action. There are three different classes of statements in C. They are

1. Expression statements
2. Compound statements
3. Control statements

Expression Statements

An expression statement consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

Example:

```
a = 3;
c = a + b;
++i;
printf("Area = %f",area);
;
```

Compound Statements

A compound statement consists of several individual statements enclosed within a pair of braces { }. The individual statements may themselves be expression statements, compound statements or control statements. A compound statement does not end with a semicolon.

Example:

```
{
        pi=3.14;
        c = 2 * pi *r;
        a = pi * r * r;
}
```

Control Statements

Control statements are used to create special program features, such as logical tests, loops and branches.

Example:

```
While (i<=n)
   {
        printf("%d\n",i);
        i++;
   }
```

SYMBOLIC CONSTANTS

A symbolic constant is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character constant or a string constant. A symbolic constant is defined as follows

```
#define symbolic-name text
```

Example:

```
#define STRENGTH  100
#define PASS_MARK 50
#define MAX  500
#define PI 3.14159
```

Symbolic names are sometimes called constant identifiers. The rules for defining symbolic constants are

1. Symbolic names have the same form as variable names.
2. No blank space between the pound sign '#' and the word define is permitted.
3. '#' must be the first character in the line.
4. A blank space is required between #define and symbolic name and between the symbolic name and the constant.
5. #define statement must not end with a semicolon.
6. After defining, the symbolic name should not be assigned any other value within the program by using an assignment statement.
7. Symbolic names are NOT declared for data types.
8. #define statements may appear anywhere in the program but before it is referenced in the program.

OPERATORS AND EXPRESSIONS

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. C operators can be classified into a number of categories. They include

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators.

Arithmetic Operators

C provides all the basic arithmetic operators. They are listed in the following table

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Example:

```
a + b
a - b
a * b
a / b
a % b
```

Here a and b are variables and are known as operands.

Integer Arithmetic

When both the operands in a single arithmetic expression are integers, the expression is called an integer expression, and the operation is called integer arithmetic. Integer arithmetic always yields an integer value.

Example, if a=14 and b=4 then

```
a + b = 18
a – b = 10
a * b = 56
a / b = 3
a % b = 2
```

Example program

```
#include<stdio.h>
main()
 {
        int a,b,c,d,e;
        printf(“Enter the numbers “);
        scanf(“%d%d”,&a,&b);
        c=a+b;
        d=a-b;
        e=a*b;
        f=a/b;
        g=a%b;
        printf(“\na+b=%d”,c);
        printf(“\na-b=%d”,d);
        printf(“\na*b=%d”,e);
        printf(“\na/b=%d”,f);
        printf(“\na%b=%d”,g);
 }
```

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation.

Example: if x, y, and z are floats, then

```
x=6.0/7.0     = 0.857143
y=1.0/3.0     = 0.333333
z=-2.0/3.0    = -0.666667
```

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression.

Example:

```
15/10.0 = 1.5
```

Relational Operators

The relational operators are used to compare two values. An expression containing a relational operator is termed as a relational expression. The value of a relational expression is either one or zero. C supports six relational operators. These operators and their meanings are shown in the following table

Operator	Meaning
<	Is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to
==	Is equal to
!=	Is not equal to

== and != relational operators are also known as equality operators. A simple relational expression contains only one relational operator and takes the following form

expression1 relational-operator expression2

Example1:

Suppose that i, j and k are integer variables whose values are 1,2 and 3, respectively. Several relational expressions involving these variables are shown below.

Expression	Interpretation	Value
i<j	true	1
(i+j)>=k	true	1
(j+k)>(i+5)	false	0
k!=3	false	0
j==2	true	1

Example2:

Suppose that i is an integer variable whose value is 7, f is a floating-point variable whose value is 5.5, and c is a character variable that represents the character ‘w’. Several relational expressions involving these variables are shown below.

Expression	Interpretation	Value
f>5	true	1
(i+f)<=10	false	0
c==119	true	1
c!=’p’	true	1
c>=10 * (i+f)	false	0

Logical Operators

In addition to the relational operators, C has three logical operators. They are

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

The logical operators && and || are used when we want to test more than one condition and make decisions. An example is

a>b && x==10

An expression of this kind which combines two or more relational expressions is termed as a logical expression or a compound relational expression. The truth table of logical &&,|| and operators are

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

A	!A
0	1
1	0

Example:

Suppose that I is an integer variable whose value is 7, f is a floating-point variable whose value is 5.5, and c is a character variable that represents the character ‘w’. Several complex logical expressions that make use of these variables are shown below.

Expression	Interpretation	Value
(i>=6) && (c ==’w’)	true	1
(i>=6) && (c ==119)	true	1
(f<11) && (i>100)	false	0
(c!=’p’)||((i+f)<=10)	true	1

Assignment Operators

Assignment operators are used to assign the result of an expression to a variable. The assignment operator is ‘=’. Assignment expressions that make use of this operator are written in the form

identifier = expression;

Where identifier generally represents a variable and expression represents a constant, a variable or a more complex expression.

Example:

```
a = 3;
x = y;
delta = 0.001;
area = 3.14 * r * r;
sum = a+b;
```

In addition, C has a set of ‘shorthand’ assignment operators. They are

```
+=
-=
*=
/=
%=
```

The general form of shorthand assignment operator is

Identifier op = expression;

Example:

Statement with simple assignment operator	Statement with shorthand operator
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

It is also possible to use multiple assignment operators in a C statement and of the form

Identifier1 = identifier2 = … = expression;

Example: Suppose that I and j are integer variables. The multiple assignment expression

i=j=10;

will cause the integer value 10 to be assigned to both i and j.

Increment and Decrement Operators

The increment and decrement operators are

++ and --

The increment operators causes its operand to be increased by 1, whereas the decrement operator causes its operand to be decreased by 1.

The increment and decrement operators can each be utilized two different ways, depending on whether the operator is written before or after the operand. If the operator precedes the operand (++i), then it is called as pre-increment operator and the operand will be altered in value before it is utilized for its intended purpose within the program. If, however, the operator follows the operand (i++), then it is known as post-increment operator and the value of the operand will be altered after it is utilized.

Example:
```
#include<stdio.h>
main()
 {
        int i=10;
        printf("\n%d",i++);
        printf("\n%d",++i);
        printf("\n%d",i--);
        printf("\n%d",--i);
 }
```

Conditional Operator (Ternary Operator)

A ternary operator pair “?:” is available in C to construct conditional expressions of the form

exp1 ? exp2 : exp2;

Where exp1, exp2, and exp3 are expressions.

Here exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

Example:

```
#include<stdio.h>
main()
 {
	int a,b,c;
	printf(“\nEnter the numbers”);
	scanf(“%d%d”,&a,&b);
	c=(a>b)?a:b;
	printf(“\nThe greatest value is %d”,c);
 }
```

Bitwise Operators

The bitwise operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. The following table lists the bitwise operators and their meanings.

Operator	**Meaning**
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right
~	One’s complement

Special Operators

C supports some special operators. They are

- comma operator
- sizeof operator
- pointer operators (& and *)
- member selection operator (. and ->)

The comma operator

The comma operator can be used to link the related expressions together.

Examples:

1. The statement

```
value = (x = 10, y=5, x+y);
```

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 to value.

2. In for loops: `for(n=1,m=10;n<=m;n++,m++)`

3. In while loops: `while(c=getchar(),c!=’\0’)`

4. Exchanging values: `t=x,x=y,y=t;`

The sizeof operator

The sizeof is a compile time operator and returns the number of bytes the operand occupies.

Example:

```
        m=sizeof(sum);
        n=sizeof(long int);
        k=sizeof(234L);
```

The sizeof operator is normally used to determine the lengths of arrays and structures.

```
#include<stdio.h>
main()
 {
        int i;
        float f;
        double d;
        char c;

        printf("\nInt : %d",sizeof i);
        printf("\nFloat : %d",sizeof f);
        printf("\nDouble : %d",sizeof(d));
        printf("\nChar : %d",sizeof(c));

        printf("\nInt : %d",sizeof int);
        printf("\nFloat : %d",sizeof float);
        printf("\nDouble : %d",sizeof(double));
        printf("\nChar : %d",sizeof(char));
 }
```

MATHEMATICAL FUNCTIONS (LIBRARY FUNCTIONS)

Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems. Most of the C compilers support these basic math functions. Some standard math functions are given in the following table.

Function	**Meaning**
acos(x)	Arc cosine of x
asin(x)	Arc sin of x
atan(x)	Arc tangent of x
atan2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the power x (e^x)
fabs(x)	Absolute value of x
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y

log(x)	Natural log of x, x>0
log10(x)	Base 10 log of x, x>0
pow(x,y)	x to the power y (x^y)
sqrt(x)	Square root of x, x>0

Note: 1. x and y should be declared as double.
2. In trigonometric and hyperbolic functions, x and y are in radians.
3. All the functions return a double.

OPERATOR PRECEDENCE AND ASSOCIATIVITY

Each operator in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of the levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as the associativity property of an operator. The following table provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence.

OPERATOR	DESCRIPTION	ASSOCIATIVITY	RANK
() []	Function call Array element reference	Left to right	1
+ - ++ -- ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer reference Address Size of an object Type cast	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ -	Addition Subtraction	Left to right	4
<< >>	Left shift Right shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== !=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9

|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

INPUT AND OUTPUT STATEMENTS

READING A CHARACTER

Reading a single character can be done by using the function getchar. The getchar takes the following form:

variable_name = getchar();

variable_name is a valid C name that has been declared as char type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to getchar function.

For example

```
char name;
name=getchar();
```

will assign the character 'H' to the variable name when we press the key H on the keyboard.

Example:

```
#include<stdio.h>
main()
 {
   char c;
   printf("Enter the character : ");
   c=getchar();
   printf("The character is %c ",c);
 }
```

WRITING A CHARACTER

Printing a single character can be done by using the function putchar. The putchar takes the following form:

putchar(variable_name);

where variable_name is a type char variable containing a character. This statement displays the character contained in the variable_name at the terminal. For example, the statements

c='Y';
putchar(c);
will display the character Y on the screen. The statement
putchar('\n');

would cause the cursor on the screen to move to the beginning of the next line.

Example program:

```
#include<stdio.h>
main()
 {
  char c;
  printf("Enter the character : ");
  c=getchar();
  printf("The character is ");
  putchar(c);
 }
```

FORMATTED INPUT

The scanf function can be used to read formatted input. The general form of scanf is

scanf("control string",arg1,arg2,…,arg-n);

The control string specifies the field format in which the data is to be entered and the arguments arg1,arg2,…,arg-n specify the address of locations where the data is stored. Control string and arguments are separated by commas.

Control string contains field specifications which direct the interpretation of input data. It may include:

- field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an optional number, specifying the field width.
- Blanks, tabs, or newlines.

Inputting Integer Numbers

The field specification for reading an integer number is:

%wd

The percent sign(%) indicated that a conversion specification follows. w is an integer number that specifies the field width of the number to be read and d, known as data type character, indicated that the number to be read is in integer mode. Consider the following example:

scanf("%2d%5d",&n1,&n2);

Data line:

10 12345

The value 10 is assigned to n1 and 12345 to n2. Suppose the input data is as follows:

12345 10

The variable n1 will be assigned 12 and n2 will be assigned 345. The value 10 that is unread will be assigned to the first variable in the next scanf call. The statement

scanf("%d%d",&n1,&n2);

will read the data

12345 10

correctly and assign 12345 to n1 and10 to n2.

An input field may be skipped by specifying ***** in the place of field width. For example, the statement

scanf("%d%*d%d",&a,&b);

will assign the data

123 456 789

as follows:

123 to a
456 skipped (because of *)
789 to b

Input Real Numbers

scanf reads real numbers using the simple specification %f for both decimal point notation and exponential notation.

Example:

scanf("%f%f%f",&x,&y,&z);

with the input data

123.45 12.21e-1 456

will assign the value 123.45 to x, 1.221 to y, and 456.0 to z.

Inputting Character Strings

A scanf function can input strings containing more than one character. Following are the specifications for reading character strings:

%ws or **%wc**

%c may be used to read a single character when the argument is a pointer to a char variable.

Reading Mixed Data Types

It is possible to use one scanf statement to input a data line containing mixed mode data. The statement

scanf("%d%c%f%s",&count,&code,&ratio,name);

will read the data

15 a 1.234 coffee

correctly and assign the values to the variables in the order in which they appear.

Commonly used scanf format codes are given in the following table.

CODE	MEANING
%c	Read a single character
%d	Read a decimal integer
%e	Read a floating point value
%f	Read a floating point value
%g	Read a floating point value
%h	Read a short integer
%i	Read a decimal, hexadecimal, or octal integer
%o	Read an octal integer
%s	Read a string
%u	Read an unsigned decimal integer
%x	Read a hexadecimal integer

The following letters may be used as prefix for certain conversion characters.

h	for short integers
l	for long integers or double
L	for long double

FORMATTED OUTPUT

The use of printf function for printing captions and numerical results. The printf function provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals. The general for of printf statement is

printf("control string",arg1,arg2,…,arg-n);

control string consists of three types of items:

1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. Escape sequence characters such as \n, \t, and \b.

The control string indicates how many arguments follow and what their types are. The arguments arg1,arg2,…,arg-n are the variables whose values are formatted and printed according to the specifications of the control string.

A simple format specification has the following form:

%w.p type-specifier

where w is an integer number that specifies the total number of columns for the output value and p is another integer number that specifies the number of digits to the right of the decimal point or the number of characters to be printed from a string. Some examples of printf statement are

```
printf("Programming in C");
printf(" ");
printf("\n");
printf("%d",x);
printf("a=%f\nb=%f",a,b);
printf("sum = %d",1234);
printf("\n\n");
```

Output of Integer Numbers

The format specification for printing an integer number is

$$\boxed{\%wd}$$

where w specifies the minimum field width for the output. d specifies that the value to be printed is an integer.

Example:

printf("%d",1234);	1	2	3	4		
printf("%6d",1234);			1	2	3	4
printf("%2d",1234);	1	2	3	4		
printf("%-6d",1234);	1	2	3	4		
printf("%06d",1234);	0	0	1	2	3	4

It is possible to force the printing to be left-justified by placing a minus sign directly after the % character as shown in the fourth example above. It is also possible to pad with zeros the leading blanks by placing a 0 before the field width specifier as shown in the last item above.

Example program:

```
#include<stdio.h>
main()
 {
   int m=1234;
   long n=987654;
   printf("\n%d",m);
   printf("\n%10d",m);
   printf("\n%010d",m);
   printf("\n%-10d",m);
   printf("\n%10d",n);
   printf("\n%10d",-n);
 }
```

Output of Real numbers

The output of a real number may be displayed in decimal notation using the following format specification:

%w.p f

The integer w indicates the minimum number of positions. The integer p indicates the number of digits to be displayed after the decimal point.

We can also display a real number in exponential notation by using the specification

%w.p e

The following examples illustrate the output of the number y=98.7654 under different format specifications:

printf("%7.4f",y);	9	8	.	7	6	5	4					
printf("%7.2f",y);			9	8	.	7	7					
printf("%-7.2f",y);	9	8	.	7	7							
printf("%f",y);	9	8	.	7	6	5	4					
printf("%10.2e",y);			9	.	8	8	e	+	0	1		
printf("%11.4e",-y);	-	9	.	8	7	6	5	e	+	0	1	
printf("%-10.2e",y);	9	.	8	8	e	+	0	1				
printf("%e",y);	9	.	8	7	6	5	4	0	e	+	0	1

Some systems also support a special field specification character that lets the user define the field size at run-time. This takes the following form:

printf("%*.*f", width, precision, number);

Example:

printf("%*.*f",7,2,no);

is equivalent to

printf("%7.2f",no);

Example program:

```
#include<stdio.h>
main()
 {
  float y=98.7654;
  printf("\n%7.4f",y);
  printf("\n%7.2f",y);
  printf("\n%-7.2f",y);
  printf("\n%f",y);
  printf("\n%10.2e",y);
  printf("\n%11.4e",y);
  printf("\n%-10.2e",y);
  printf("\n%e",y);
 }
```

Printing of a Single Character

A single character can be displayed in a desired position using the format

%wc

The character will be displayed right-justified in the field of w columns. We can make the display left-justified by placing a minus sign before the integer w.

Printing of Strings

The format specification for outputting strings is similar to that of real numbers. It is of the form

%w.ps

where w specifies the field width and p instructs that only the first p characters of the string are to be displayed. The display is right-justified.

The following examples show the effect of a variety of specifications in printing a string "**NEW DELHI 110001**", containing 16 characters (including space).

%s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				
%20s					N	E	W		D	E	L	H	I		1	1	0	0	0	1
%20.10s											N	E	W		D	E	L	H	I	
%.5s	N	E	W		D															
%-20.10s	N	E	W		D	E	L	H	I											
%5s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				

Mixed Data Output

It is permitted to mix data types in one printf statement. For example,

printf("%d %f %s %c", a, b, c, d);

printf uses its control string to decide how many variables to be printed and what their types are. Therefore, the format specifications should match the variables in number, order, and type.

Example program:

```
#include<stdio.h>
main()
 {
   char name[20]="ANIL KUMAR GUPTA";
   printf("\n%s",name);
   printf("\n%20s",name);
   printf("\n%20.10s",name);
   printf("\n%-20.10s",name);
   printf("\n%5s",name);
   printf("\n%.5s",name);
 }
```

Commonly used printf format codes are given in the following table

CODE	**MEANING**
%c	Print a single character
%d	Print a decimal integer
%e	Print a floating point value in exponent form
%f	Print a floating point value without exponent
%g	Print a floating point value either e-type or f-type depending on value
%i	Print a signed decimal integer
%o	Print an octal integer, without leading zero
%s	Print a string
%u	Print an unsigned decimal integer
%x	Print a hexadecimal integer, without leading 0x

The following letters may be used as prefix for certain conversion characters.

h	for short integers
l	for long integers or double
L	for long double

The format flags are shown in the following table

FLAG	**MEANING**
-	Output is left-justified within the field. Remaining field will be blank
+	+ or – will precede the signed numeric item.
0	Causes leading zeroes to appear.
# (with o or x)	Causes octal and hex items to e preceded by 0 or 0x, respectively
# (with e, f or g)	Causes a decimal point to be present in all floating point numbers, even if it is while number. Also prevents the truncation of trailing zeros in g-type conversion.

DECISION MAKING AND BRANCHING

C language supports control or decision making statements. They are

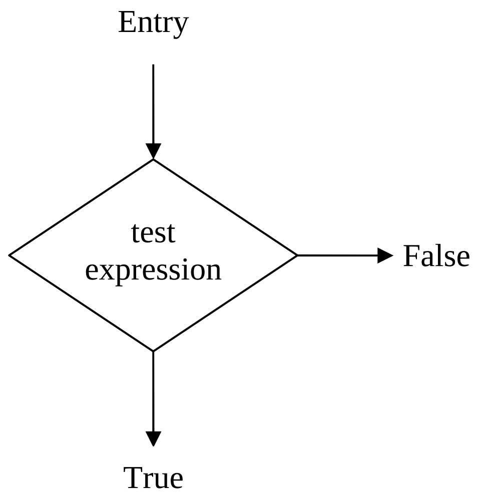
1. if statement
2. switch statement
3. Conditional operator statement
4. goto statement

Decision making with if statement

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. The syntax of if statement is

```
If (condition)
```

It allows the computer to evaluate the condition first and then, depending on whether the value of the condition is 'true' (non-zero) or 'false' (zero), it transfers the control to a particular statement. This point of program has two paths to follow, one for the true condition and the other for the false condition as shown in the following fig.



Types of if statements

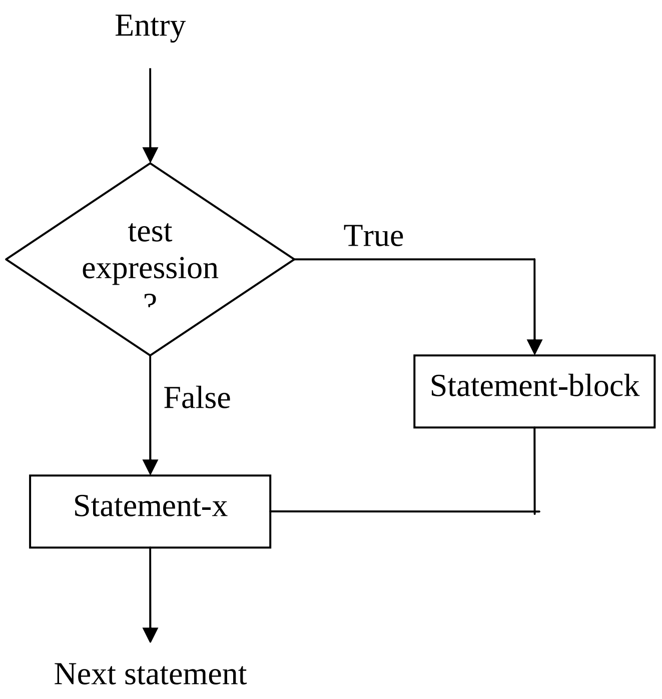
1. Simple if statement
2. if….else statement
3. Nested if….else statement
4. else if ladder

Simple if statement

The general form of a simple if statement is

```
if (condition)
  {
    Statement-block;
  }
Statement-x;
```

The statement-block may be single statement or a group of statements. If the condition is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x. The flowchart of simple if statement is



Example:

```
#include<stdio.h>
#include<conio.h>
void main()
  {
    int a,b;
    clrscr();
    printf(“Enter the numbers : “);
    scanf(“%d%d”,&a,&b);
    if (a>b)
      printf(“A is greatest”);
    if (b>a)
      printf(“B is greatest”);
    getch();
  }
```

The if…else statement

The if…else statement is an extension of the simple if statement. The general form is

```
if (condition)
  {
    True-block Statement(s);
  }
else
  {
    False-block Statement(s);
  }
 Statement-x;
```

If the condition is true, then the true-block statement(s), immediately following the if statement are executed; otherwise, the false-block statement(s) are executed. The flowchart of if…else statement is

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
  {
    int a,b;
    clrscr();
    printf("Enter the numbers : ");
    scanf("%d%d",&a,&b);
    if (a>b)
      printf("A is greatest");
    else
      printf("B is greatest");
    getch();
  }
```

Nesting of if…else statements

When a series of decisions are involved, we may have to use more than one if…else statement in nested form as follows:

```
if (condition 1)
  {
    if (condition 2)
      {
        Statement-1;
      }
    else
      {
        Statement-2;
      }
  }
else
  {
    Statement-3;
  }
 Statement-x;
```

If the condition-1 is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the condition-2 is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x. The flowchart of nested if…else statement is

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
  {
    int a,b,c;
    clrscr();
    printf("Enter the numbers : ");
    scanf("%d%d%d",&a,&b,&c);
    if (a>b)
      {if (a>c)
          printf("A is greatest");
       else
          printf("C is greatest");
      }
    else
      {if (b>c)
          printf("B is greatest");
       else
          printf("C is greatest");
      }
    getch();
  }
```

The else…if ladder (if…else if statement)

A multipath decision is a chain of ifs in which the statement associated with each else is an if. The general form of if…else if statement is

```
if (condition 1)
  Statement-1;
else if (condition 2)
      Statement-2;
     else if (condition 3)
           Statement-3;
            …………
          else if (condition n)
               Statement-n;
              else
                Default-Statement;
Statement-x;
```

This construct is known as the else if ladder. The conditions are evaluated from the top, downwards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x. When all the n conditions become false, then the final else containing the default-statement will be executed. The following shows the logic of execution of else if ladder statements.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
  {
    int a,b,c;
    clrscr();
    printf(“Enter the numbers : “);
    scanf(“%d%d%d”,&a,&b,&c);
    if ((a>b)&&(a>c))
          printf(“A is greatest”);
    else if (b>c)
               printf(“B is greatest”);
          else
               printf(“C is greatest”);
    getch();
  }
```

The switch statement

The switch statement is a built-in multiway decision statement. The switch statement tests the value of a given variable against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below:

```
switch(expression)
 {
   case value-1:
                    block-1;
                    break;
   case value-2:
                    block-2;
                    break;
     ……
     ……
     ……
   case value-n:
                    block-n;
                    break;
   default:
                    default-block;
                    break;
 }
```

The expression is an integer expression or characters. Value-1, value-2,… are constants or constant expressions and are known as case labels. Each of these values should be unique within a switch statement. Block-1, block-2,… are statement lists and may contain zero or more statements.

When the switch is executed, the value of the expression is successively compared against the values value-1, value-2,… if a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. The flowchart of switch statement is

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
 {
  int a,b,ch;
  float c;
  clrscr();
  printf("Enter the numbers\n");
  scanf("%d%d",&a,&b);
  printf("\n1.Addition");
  printf("\n2.Subtraction");
  printf("\n3.Multiplication");
  printf("\n4.Division");
  printf("\nEnter your choice : ");
  scanf("%d",&ch);
switch(ch)
 {
  case 1:c=a+b;
           break;
  case 2:c=a-b;
           break;
  case 3:c=a*b;
           break;
  case 4:c=a/b;
           break;
  default:printf("Invalid Choice");
           getch();
           exit(0);
           break;
 }
printf("Result is %f",c);
getch();
}
```

The ?: Operator(Ternary Operator/Conditional Operator)

The ?: is a two way decision making statement. This operator is a combination of ? and :, and takes three operands. This operator is popularly known as the conditional operator. The general form of use of the conditional operator is as follows:

conditional expression ? expression1 : expression2;

The conditional expression is evaluated first. If the result is nonzero, expression1 is evaluated and is returned as the value of the conditional expression. Otherwise, expression2 is evaluated and its value is returned. For example, the segment

```
if (x<0)
    flag=0;
else
    flag=1;
```

can be written as

```
flag=(x<0)?0:1;
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
  {
   int a,b,c;
   clrscr();
   printf("Enter the numbers : ");
   scanf("%d%d",&a,&b);
   c=(a>b)?a:b;
   printf("%d is greatest",c);
   getch();
  }
```

The goto statement

The goto statement is used to branch unconditionally from one point to another in the program. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred. The general forms of goto and label statements are shown below:

```
goto label;
    …
    …
    …
label:
Statement;
```

Forward jump

```
label:
Statement;
    …
    …
    …
goto label;
```

Backward jump

The label: can be anywhere in the program either before or after the goto label; statement. If the label: is before the statement goto label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a backward jump. On the other hand, if the label: is placed after the goto label; some statements will be skipped and the jump is known as a forward jump.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
 {
  int i=1;
  clrscr();
  first:
    printf("\n%d",i);
    i++;
    if (i<=10) goto first;
  getch();
 }
```

DECISION MAKING AND LOOPING

Loop is a sequence of statements executed until some conditions for termination of the loop are satisfied. A program loop consists of two segments, one known as the body of the loop and the other known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the entry-controlled loop or as the exit-controlled loop. The following flowchart shows these structure.

A looping process would include the following four steps

1. Setting and initialization of a counter.
2. Execution of the statements in the loop.
3. Test for a specified condition for execution of the loop.
4. Incrementing the counter.

The C language provides for three loop constructs for performing loop operations. They are:

1. The while statement.
2. The do statement
3. The for statement.

The while Statement

The simplest of all the looping structures in C is the while statement. The while is an entry-controlled loop statement. The condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop. The basic format of the while statement is

```
while (condition)
 {
   body of the loop
 }
```

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
  {
   int i=1;
   clrscr();
   while(i<=10)
     {
       printf("\n%d",i);
       i++;
     }
   getch();
  }
```

The do…while statement

In do…while statement, the body of the loop is evaluated first. At the end of the loop, the condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement. The general format of the do…while statement is

```
do
  {
    body of the loop
   } while (condition);
```

Since the condition is evaluated at the bottom of the loop, the do…while construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
  {
   int i=1;
   clrscr();
  do
    {
       printf(“\n%d“,i);
       i++;
    } while(i<=10);
  getch();
}
```

The for statement
Simple for loops
The for loop is another entry-controlled loop that provides a more concise loop control structure. The general form of the for loop is

```
for(initialization; condition; increment)
  {
        body of the loop
  }
```

The execution of the for statement is as follows:

1. Initialization of the control variables is done first. Eg. i=1 and count=0.
2. The value of the control variable is tested using the condition. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the increment section is executed and the condition is evaluated. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the condition.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
 {
   int i;
   clrscr();
   for(i=1;i<=10;i++)
     {
          printf(“\n%d“,i);
     }
   getch();
 }
```

Nesting of for loops
One for statement within another for statement is known as nested for loops. The general form of nested for loop is

```
for(initialization; test-condition; increment)
  {
      for(initialization; test-condition; increment)
        {
            body of the loop
        }
  }
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
 {
  int i,j;
  clrscr();
  for(i=1;i<=7;i++)
    {
       for(j=1;j<i;j++)
         {
            printf("%d ",j);
         }
       printf("\n");
    } getch();
 }
```

Additional Features of for Loop

- More than one variable can be initialized at a time in the for statement.

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

- The increment section may also have more than one part

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

- The test-condition may have any compound relation

```
for(i=0,j=1;i<n && j< n;i+=2,j+=2)
```

```
   {
         printf("\n%d\t%d",i,j);
   }
```

- Expressions in the assignment statements of initialization and increment section

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

- One or more sections can be omitted

```
x=5;
for(;x<=100;)
   {
         ……
         ……
         ……
   }
```

- We can set up time delay loops using the null statement

```
for(x=1000;x>0;x--)
         ;
         or
for(x=1000;x>0;x--);
```

Jumps In Loops

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.

Jumping Out of a Loop

An early exit from a loop can be accomplished by using the break statement or the goto statement. These statements can also be used within while, do, or for loops. The general form of break statement is

```
break;
```

When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

Example 1:

```
while(condition)
 {
      …
      …
      if (condition)
        break;
      …
      …
 }
```

Example 2:

```
do
 {
      …
      …
      if (condition)
        break;
      …
      …
 } while(condition);
```

Example 3:

```
for(……)
 {
      …
      …
      if (condition)
        break;
      …
      …
 }
```

Example 4:

```
for(……)
 {
      …
      …
      for(……)
       {
            …
            …
            if (condition)
              break;
            …
            …
      }
      …
      …
 }
```

Example Program:

```
#include<stdio.h>
#include<conio.h>
void main()
  {
    int i=1;
    clrscr();
    for(;;)
      {
        printf("\n%d",i);
        i++;
        if (i==10) break;
      }
    getch();
  }
```

Skipping a part of a loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions.

The continue statement causes the loop to be continued with the next iteration after skipping any statements in between. The general form of continue statement is

```
continue;
```

Example 1:

```
while(condition)
  {
        …
        …
        if (condition)
          continue;
        …
        …
  }
```

Example 2:

```
do
  {
        …
        …
        if (condition)
          continue;
        …
        …
  } while(condition);
```

Example 3:

```
for(……)
 {
      …
      …
      if (condition)
        continue;
      …
      …
 }
```

Example 4:

```
for(……)
 {
      …
      …
      for(……)
        {
            …
            …
            if (condition)
              continue;
            …
            …
      }
      …
      …
 }
```

Example Program:

```
#include<stdio.h>
#include<conio.h>
void main()
 {
   int i;
   clrscr();
   for(i=1;i<100;i++)
     {
        if ((i%2)==0) continue;
        printf("\n%d",i);
     }
   getch();
 }
```

ARRAYS

An array is a group of related data items that share a common name. For example,

a[10]

represents any 10 values. While the complete set of values is referred to as an array, the individual values are called elements.

One-Dimensional Arrays

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array. The subscript can begin with number 0. That is a[0] is allowed. For example, if we want to represent a set of five numbers by an array variable number, then we may declare the variable no as follows

int no[5];

and the computer reserves five storage locations as shown below:

	no[0]
	no[1]
	no[2]
	no[3]
	no[4]

The values to the array elements can be assigned as follows:

no[0]=37
no[1]=98
no[2]=57
no[3]=35
no[4]=71

This would cause the array no to store the values as shown below:

37	**no[0]**
98	**no[1]**
57	**no[2]**
35	**no[3]**
71	**no[4]**

These elements may be used in programs like any other C variable.

Example:

```
a=no[0]+25;
no[3]=no[1]+no[2];
a[7]=no[3]*7;
```

Declaration of Arrays

Arrays must be declared before they are used. The general form of array declaration is

type variable-name[size];

The type specifies the type of element that will be contained in the array, such as int, float, or char and the size indicates the maximum number of elements that can be stored inside the array. For example,

float height[25];

declares the height to be an array containing 25 real elements. Any subscripts 0 to 24 are valid.

Initialization of Arrays

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

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

The values in the list are separated by commas. For example, the statement

static int no[3]={0,0,0};

will declare the variable no as an array of size 3 and will assign zero to each element.

The size may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, the statement

static int counter[]={1,1,1,1};

will declare the counter array to contain four elements with initial values 1.

Character arrays may be initialized in a similar manner. Thus, the statement

static char name[]={‘J’,’O’,’H’,’N’,’\0’};

declares the name to be an array of four characters, initialized with the string “JOHN”.

Initialization of arrays in C suffers two drawbacks.

1. There is no convenient way to initialize only selected elements.
2. There is no shortcut method for initializing a large number of array elements.

Example Program:

```
#include<stdio.h>
#include<conio.h>
main()
 {
  int a[10],i,s=0,n;
  clrscr();
  printf(“Enter the value of n:”);
  scanf(“%d”,&n);
  printf("Enter the numbers : );
  for(i=0;i<n;i++)
   scanf("%d",&a[i]);
  for(i=0;i<n;i++)
   s+=a[i];
  printf("The sum is %d",s);
  getch();
}
```

Two-Dimensional Arrays

C allows us to define a table of items by using two-dimensional arrays. The two-dimensional arrays are declared as follows:

type array_name[row_size][column_size];

Two-dimensional arrays are stored in memory as shown in the following fig.

	Column 0	**Column 1**	**Column 2**
Row 0	**0,0**	**0,1**	**0,2**
Row 1	**1,0**	**1,1**	**1,2**
Row 2	**2,0**	**2,1**	**2,2**
Row 3	**3,0**	**3,1**	**3,2**

Each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

Example Program:

```
#include<stdio.h>
#include<conio.h>
main()
 {
  int a[10][10],b[10][10],c[10][10],i,j,m,n;
  clrscr();
  printf("Enter the order of the matrix");
  scanf("%d%d",&m,&n);
  printf("Enter the first matrix : );
  for(i=0;i<m;i++)
    for(j=0;j<n;j++)
      scanf("%d",&a[i][j]);
  printf("Enter the second matrix : );
  for(i=0;i<m;i++)
    for(j=0;j<n;j++)
      scanf(“%d”,&b[i][j]);
  for(i=0;i<m;i++)
    for(j=0;j<n;j++)
      c[i][j]=a[i][j]+b[i][j];
  printf(“The resultant matrix : );
  for(i=0;i<m;i++)
    {
      for(j=0;j<n;j++)
       printf(“%d\t”,&c[i][j]);
      printf(“\n”);
    }
  getch();
 }
```

Initializing Two-Dimensional Arrays

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
static int table[2][3] = {0,0,0,1,1,1};
```

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be written as

```
static int table[2][3] = {{0,0,0},{1,1,1}};
```

We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
static int table[2][3] = {
                          {0,0,0},
                          {1,1,1}
                         };
```

If the values are missing in an initializer, they are automatically set to zero. For instance, the statement

```
static int table[2][3] = {
                          {1,1},
                          {2}
                         };
```

will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.

When all the elements are to be initialized to zero, the following short-cut method may be used.

```
static int table[2][3] = {
                          {0},
                          {0}
                         };
```

Multidimensional arrays

C allows arrays of three or more dimensions. The general form of a multidimensional array is

type array_name[s1][s2][s3]…[sn];

where s_i is the size of the ith dimension. Some example are:

```
int survey[3][5[12];
float table[5][4][5][3];
```

survey is a three-dimensional array declared to contain 180 integer type elements. Similarly table is a four-dimensional array containing 300 elements of floating-point type.

HANDLING OF CHARACTER STRINGS

Introduction

A string is an array of characters. Any group of characters defined between double quotation marks is a constant string.

Example:

"Welcome to C"

Character strings are often used to build meaningful and readable programs. The common operations performed on character strings are:

- Reading and Writing strings
- Combining strings together.
- Copying one string to another
- Comparing strings for equality
- Extracting a portion of a string.

Declaring and Initializing String Variables

The general form of declaration of a string variable is

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

The size determines the number of characters in the string-name.
Example:

```
char name[30];
char city[20];
```

when the compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at the end of the string.

Character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
static char city[9] = "NEW YORK";
static char city[9] = {'N','E','W',' ','Y','O','R','K','\0'};
```

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized.

Example:

```
static char city[] = {'N','E','W',' ','Y','O','R','K','\0'};
```

defines the array city as a nine element array.

Reading Strings from Terminal

Reading Words

The familiar input function scanf can be used with %s format specification to read in a string of characters.

Example:

```
char city[20];
scanf("%s",city);
```

The problem with the scanf function is that it terminates its input on the first white space it finds. Therefore, if the following line of text is types in at the terminal,

NEW YORK

Then only the string **"NEW"** will be read into the array city, since the blank space after the word **"NEW"** will terminate the string.

Example Program:

```
#include<stdio.h>
#include<conio.h>
main()
 {
  char name[30];
  clrscr();
  printf("Enter the Name : ");
  scanf("%s",name);
  printf("Name : %s",name);
  getch();
 }
```

Reading a Line of Text

An entire line of text can be read and stored in an array using gerchar() function. The reading is terminated when the newline character('\n') is entered and the null character is then inserted at the end of the string.

Example:

```
#include<stdio.h>
#include<conio.h>
main()
 {
  char name[30],c;
  int i=0;
  clrscr();
  printf("Enter the Name : ");
  do
    {
      c=getchar();
      name[i]=c;
      i++;
    }while (c!=’\n’);
  i--;
  name[i]=’\0’;
  printf(“Name : %s”,name);
  getch();
}
```

Writing Strings to Screen

The printf function with %s format is used to print strings to the screen. The format %s can be used to display an array of characters that is terminated by the null character.

Example:

printf(“%s”,name);

We can also specify the precision with which the array is displayed. For instance, the specification

%10.4

indicates that the first four characters are to be printed in a field width of 10 columns.

Example Program: Coping one string into another

```
#include<stdio.h>
#include<conio.h>
main()
 {
  int  s1[30],s2[30],i=0;
  clrscr();
  printf(“Enter the String : ”);
  scanf(“%s”,s1);
  do
   {
    s2[i]=s1[i];
    i++;
   }while(s1[i-1]!=’\0’);
  printf(“\nS1 : %s”,s1);
  printf(“\nS2 : %s”,s2);
  getch();
 }
```

Arithmetic Operations on Characters

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system. For example, if the machine uses the ASCII representation, then,

```
x='a';
printf("%d",x)
```

will display the number 97 on the screen.

It is also possible to perform arithmetic operations on the character constants and variables. For example,

```
x='z'-1;
```

is a valid statement. In ASCII, the value of 'z' is 122 and therefore, the statement will assign the value 121 to the variable x.
We may also use character constants in relational expressions. For example, the expression

```
ch>='A' && ch<='Z'
```

would test whether the character contained in the variable ch is an upper-case letter.

We can convert a character digit to its equivalent integer value using the following relationship:

```
x=ch-'0';
```

where x is defined as an integer variable and ch contains the character digit, For example, let us assume that the ch contains the digit '7', Then,

x= ASCII value of '7' – ASCII value of '0'
= 55 – 48
= 7

The C library supports a function that converts a string of digits into their integer values. The function takes the form

```
x=atoi(string);
```

x is an integer variable and string is a character array containing a string of digits. Consider the following segment of a program:

```
int year;
char no[]="2007";
year=atoi(no);
```

no is a string variable which is assigned the string constant “2007”. The function atoi converts the string “2007” to its numeric equivalent 2007 and assigns it to the integer variable year.

Example program:

```
#include<stdio.h>
#include<conio.h>
main()
 {
   char no[]=”2007”;
   int y;
   y=atoi(no);
   clrscr();
   printf(“\nYear : %d”,y);
   getch();
 }
```

String Handling Functions

C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations. The most commonly used string-handling functions are:

Function	Action
strcat()	Concatenates two strings
strcmp()	Compares two strings
strcpy()	Copies one string over another
strlen()	Finds the length of a string

strcat() Function

The strcat() function joins two strings together. It takes the following form:

strcat(string1,string2);

string1 and string2 are character arrays. When the function strcat is executed, string2 is appended to string1. The string at string2 remains unchanged. For example, consider the following three strings:

s1=”VERY”;
s2=”GOOD”;
strcat(s1,s2);

strcat function may also append a string constant to a string variable.

Example:

strcat(s1,”GOOD”);

C permits nesting of strcat functions. For example, the statement

strcat(strcat(s1,s2),s3);

concatenates all the three strings together. The resultant string is stored in s1.

Example Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
 {
   char s1[20],s2[20];
   clrscr();
   printf("Enter the First String : ");
   scanf("%s",s1);
   printf("Enter the Second String : ");
   scanf("%s",s2);
   strcat(s1,s2)
   printf("\ns1 : %s",s1);
   printf("\ns2 : %s",s2);
   getch();
 }
```

strcmp() Function

The strcmp() function compares two strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

strcmp(string1,string2);

string1 and string2 may be string variables or string constants.

Example:

strcmp(s1,s2);
strcmp(s1,"GOOD");
strcmp("ROM","RAM");

The statement

strcmp("their", "there");

will return a value of -9 which is the numeric difference between ASCII "i" and ASCII "r". That is "i" minus "r" in ASCII code is -9. If the value is negative, string1 is alphabetically above string2.

Example Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
```

```
main()
 {
  char s1[20],s2[20];
  int n;
  clrscr();
  printf("Enter the First String : ");
  scanf("%s",s1);
  printf("Enter the Second String : ");
  scanf("%s",s2);
  n=strcmp(s1,s2)
  if (n==0)
    printf("\n%s = %s",s1,s2);
  else if (n>0)
    printf("\n%s > %s",s1,s2);
  else
    printf("\n%s < %s",s1,s2);
  getch();
 }
```

strcpy() Function

The strcpy() function works almost like a string-assignment operator. It takes the form

$$\boxed{\textbf{strcpy(string1,string2);}}$$

and assigns the contents of string2 to string1. string2 may be a character array variable or a string constant. For example, the statement

strcpy(city,”CHENNAI”);

will assign the string “CHENNAI” to the string variable city. Similarly, the statement

strcpy(city1,city2);

will assign the contents of the string variable city2 to the string variable city1.

Example Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
 {
  char s1[20],s2[20];
  clrscr();
  printf("Enter the First String : ");
  scanf("%s",s1);
  printf("Enter the Second String : ");
  scanf("%s",s2);
  strcpy(s1,s2)
  printf("\ns1 : %s",s1);
  printf("\ns2 : %s",s2);
  getch();
 }
```

strlen() Function

The strlen() function counts and returns the number of characters in a string.

n=strlen(string);

where n is an integer variable which receives the value of the length of the string.

Example Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
 {
  char s1[20],s2[20];
  int n;
  clrscr();
  printf("Enter the First String : ");
  scanf("%s",s1);
  printf("Enter the Second String : ");
  scanf("%s",s2);
  n=strlen(s1);
  printf("\nLength of s1 : %d",n);
  printf("\nLenght of s2 : %s",strlen(s2));
  getch();
 }
```

Table of strings

A list of names can be treated as a table of strings and a two –dimensional character array can be used to store the entire list. For example, a character array student[30][11] may be used to store a list of 30 names, each of length not more than 11 characters. Shown below is a table of five cities:

C	H	A	N	D	I	G	A	R	H	
C	H	E	N	N	A	I				
A	H	M	E	D	A	B	A	D		
H	Y	D	E	R	A	B	A	D		
M	U	M	B	A	I					

This table can be conveniently stored in a character array city by using the following declarations:

```
static char city[][]={   “CHANDIGARH”,
                         “CHENNAI”,
                         “AHMEDABAD”,
                         “HYDERABAD”,
                         “MUMBAI”
                       };
```

Example Program:

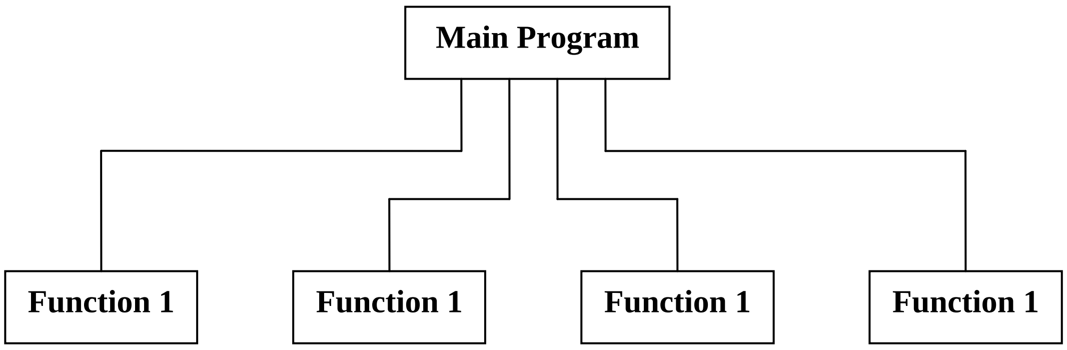
```
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
 {
   char name[30][20],t[20];
   int i,j,n;
   clrscr();
   printf(“Enter total number of names : ”);
   scanf(“%d”,&n);
   printf(“Enter the names\n”);
   for(i=0;i<n;i++)
     scanf(“%s”,name[i]);
   for(i=0;i<n;i++)
     for(j=i+1;j<n;j++)
       if (strcmp(name[i],name[j])>0)
          {
            strcpy(t,name[i]);
            strcpy(name[i],name[j]);
            strcpy(name[i],t);
          }
   for(i=0;i<n;i++)
     printf(“\n%s”,name[i]);
   getch();
 }
```

USER-DEFINED FUNCTIONS

Introduction

C functions can be classified into two categories, namely, library functions and user-defined functions. Advantages of user-defined functions are

1. It facilitates top-down modular programming as shown in the following figure.
2. The length of a 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.



A Multi-Function Program

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a 'black box' that takes some data from the main program and returns a value.

Consider a set of statements as shown below:

```
printline()
 {
  int i;
  for(i=1;i<70;i++)
     printf("-");
  printf("\n");
 }
```

The above set of statements defines a function called printline which could print a line of 69 character length. This function can be used in a program as follows:

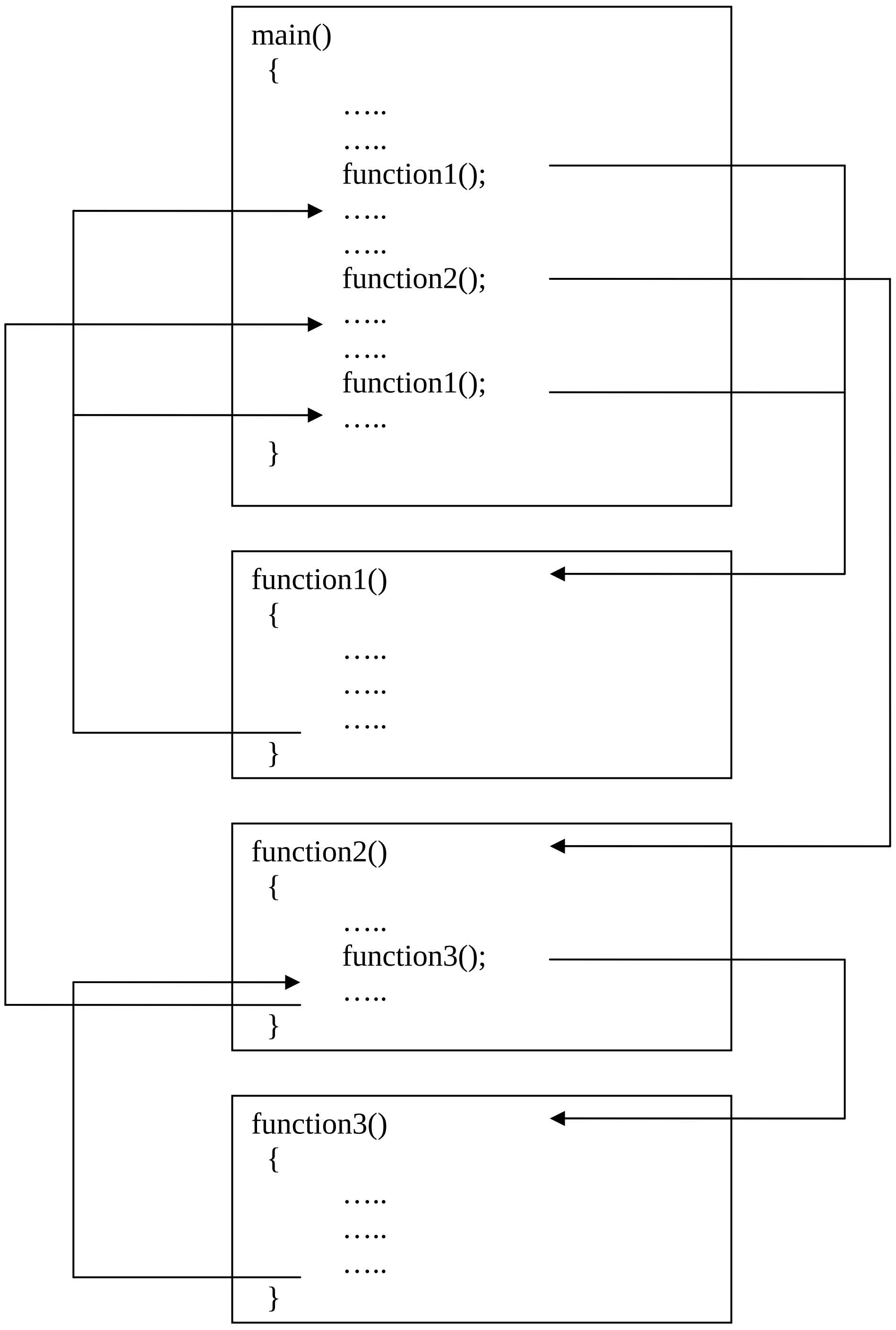
```
main()
 {
  printline();
  printf("Welcome to C");
  printline();
 }
```

```
printline()
 {
   int i;
   for(i=1;i<70;i++)
      printf("-");
   printf("\n");
 }
```

The program will print the following output:

```
---------------------------------------------------------------------
                           Welcome to C
---------------------------------------------------------------------
```

The following figure shows the flow of control in a multi-function program.



```
main()
  {
          …..
          …..
          function1();
          …..
          …..
          function2();
          …..
          …..
          function1();
          …..
  }
```

```
function1()
  {
          …..
          …..
          …..
  }
```

```
function2()
  {
          …..
          function3();
          …..
  }
```

```
function3()
  {
          …..
          …..
          …..
  }
```

The Form of C Functions

The general form of function is

```
function-name(argument list)
argument declaration
{
  local variable declarations;
  executable statement1;
  executable statement2;
        ……….
        ……….
  return(expression);
}
```

The declaration of local variables is required only when any local variables are used in the function.

The return statement is the mechanism for returning a value to the calling function.

Function Name:

A function must follow the same rules of formation as other variable names in C.

Argument List:

The argument list contains valid variable names separated by commas. The list must be surrounded by parentheses. The argument variables receive values from the calling function. Some examples of functions with arguments are:

```
quadratic(a,b,c)
square(x)
power(x,y)
fact(n)
display(res)
```

Return Values and Their Types

A function may or may not send back any value to the calling function. If it does, it is done through the return statement. The return statement can take one of the following forms

return;
Or
return(expression);

When a return is encountered, the control is immediately passes back to the calling function. An example of the use of a simple return is as follows:

```
if(error)
return;
```

The second form of return with an expression returns the value of the expression. For example, the function

```
mul(x,y)
int x,y;
{
 int p;
 p=x*y;
 return(p);
}
```

returns the value of p. The last two statements can be combined into one statement as follows:

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

A function may have more than one return statements. Example:

```
if(x<=0)
 return(0);
else
 return(1);
```

Calling a Function

A function can be called by simply using the function name in a statement. Example:

```
main()
{
  int p;
  p=mul(5,2);
  printf("%d",p);
}
```

when the compiler encounters a function call, the control is transferred to the function mul(x,y). This function is then executed line by line as described and a value is returned when a return statement is encountered. This value is assigned to p.

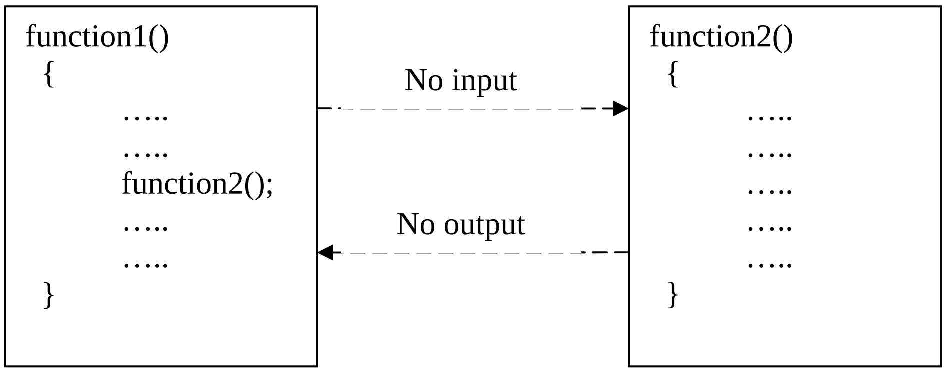
CATEGORY OF FUNCTIONS

A function may belong to one of the following categories:

Category 1: Functions with no arguments and no return values
Category 2: Functions with arguments and no return values
Category 3: Functions with arguments and return values

No Arguments and No Return Values

When a function has no arguments, there is no data transfer between the calling function and the called function. This is depicted in the following figure. The dotted lines indicate that there is only a transfer of control but not data.



Example 1:

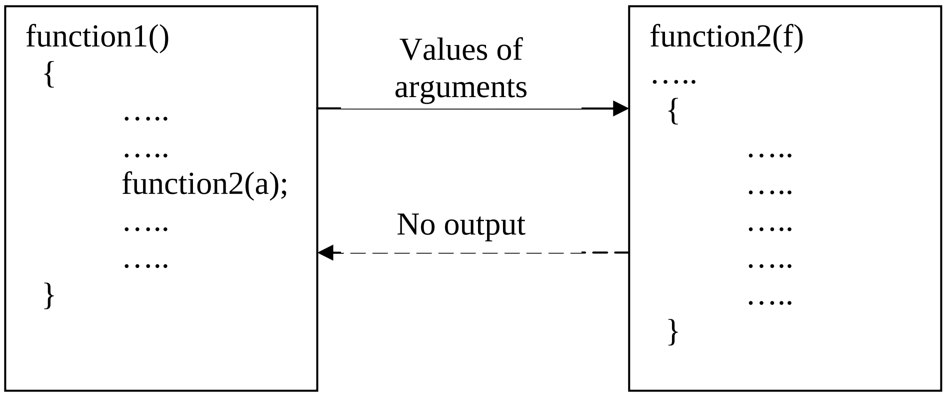
```
#include<stdio.h>
#include<conio.h>
printline()
 {
        int i;
        for(i=1;i<80;i++)
          printf("-");
        printf("\n");
 }
main()
 {
        printline();
        printf("Welcome to C");
        printline();
 }
```

Example 2:

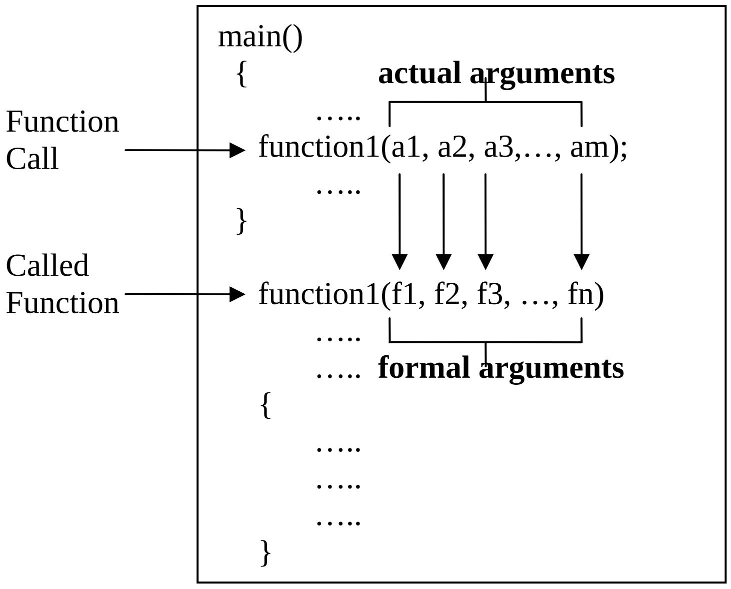
```
#include<stdio.h>
#include<conio.h>
sum()
 {
        int a,b,c;
        printf("Enter the values :");
        scanf("%d%d",&a,&b);
        c=a+b;
        printf("The sum is %d",c);
 }
main()
 {
        sum();
 }
```

Arguments But No Return Values

The nature of data communication between the calling function and the called function with arguments but no return value is shown in following figure



The arguments in the function are called formal arguments or dummy arguments. The arguments in the function call are called actual arguments. The actual and formal arguments should match in number, type, and order. The values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument as shown in following figure.



When a function call is made, only a copy of the values of actual arguments is passed into the called function. What occurs inside the function will have no effect on the variables used in the actual argument list.

Example 1:

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

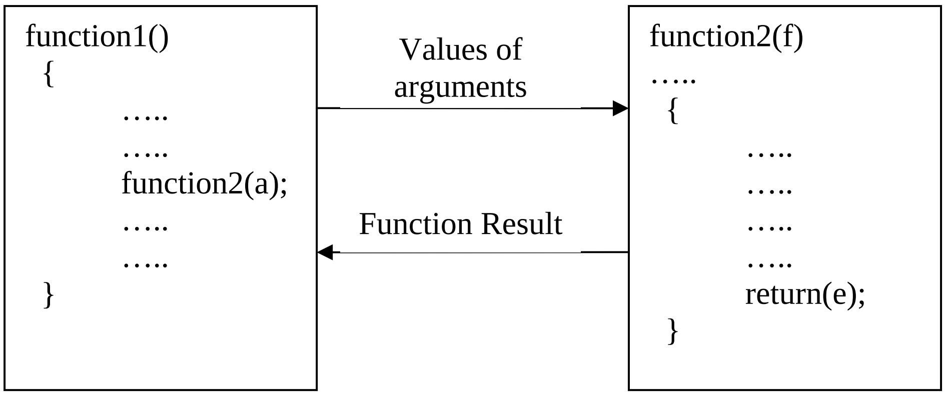
```
printline(c,n)
char c;
int n;
{
        int i;
        for(i=1;i<n;i++)
          printf("%c",c);
        printf("\n");
 }
main()
 {
        printline('-',80);
        printf("Welcome to C");
        printline('*',80);
 }
```

Example 2:

```
#include<stdio.h>
#include<conio.h>
sum(x,y)
int x,y;
 {
        int z;
        z=x+y;
        printf("The sum is %d",z);
 }
main()
 {
        int a,b;
        printf("Enter the values :");
        scanf("%d%d",&a,&b);
        sum(a,b);
 }
```

Arguments With Return Values

A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value. Such functions will have two-way data communication as shown in the following figure



C function returns a value of the type int as the default case when no other type is specified explicitly. There are two things to enable a calling function to receive a non-integer value from a called function:

1. The explicit type-specifier, corresponding to the data type required must be mentioned in the function header. The general form of the function definition is

```
type-specifier function-name(argument list)
argument declaration;
 {
   function statements;
 }
```

The type-specifier tells the compiler, the type of data the function is to return.

2. The called function must be declared at the start of the body in the calling function.

Example 1:

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

int sum(x,y)
int x,y;
 {
        int z;
        z=x+y;
        return(z);
 }

main()
 {
        int a,b,c;
        printf("Enter the values :");
        scanf("%d%d",&a,&b);
        c=sum(a,b);
        printf("The sum is %d",c);
 }
```

Functions Returning Nothing

It is possible to declare the functions with the qualifier void. This states explicitly that the functions do not return values.

Example:

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

```
void printline()
 {
        int i;
        for(i=1;i<80;i++)
          printf("-");
        printf("\n");
 }
void starline()
 {
        int i;
        for(i=1;i<80;i++)
          printf("*");
        printf("\n");
 }
main()
 {
        printline();
        printf("Welcome to C");
        starline();
 }
```

Nesting Of Functions

Calling a function with in another function is called nesting of functions.

Example: 1!+2!+3!+….+n!

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

int fact(int n)
 {
        int f=1,i;
        for(i=1;i<=n;i++)
          f*=i;
        return(f);
 }

int sum()
 {
        int s=0,i;
        for(i=1;i<=n;i++)
          s+=fact(i);
        return(s);

 }

main()
 {
        sum();
 }
```

Recursion

Calling a function with in the same function is called recursion.

Example 1:

```
#include<stdio.h>
#include<conio.h>
main()
 {
        printf("\nWelcome to C");
        main();
 }
```

When executed, this program will produce an output something like this:

```
Welcome to C
Welcome to C
Welcome to C
Welcome t
```

Execution is terminated abruptly; otherwise the execution will continue indefinitely.

Another useful example of recursion is the evaluation of factorials of a given number. The factorial of a number n is expressed as a series of repetitive multiplications as shown below.

Factorial of n = n(n-1)(n-2)(n-3)….1

For example,

Factorial of 5 = 5 X 4 X 3 X 2 X 1 = 120

Example Program:

```
#include<stdio.h>
#include<conio.h>
int fact(int n)
 {
        int f,i;
        if (n==1)
          return(1);
        else
          f=n*fact(n-1);
        return(f);
 }
main()
 {
        int n;
        printf("Enter the value of n : ");
        printf("Factorial of %d is %d",n,fact(n));
 }
```

Functions with arrays

It is also possible to pass the values of an array to a function. To pass an array to a called function, it is sufficient to list the name of the array, without any subscripts, and the size of the array as arguments. For example, the call

```
largest(a,n);
```

will pass all the elements contained in the array ***a*** of size ***n***. The largest function might look like:

```
float largest(array,size)
float array[];
int size;
 {
        ….
        ….
        ….
 }
```

The function largest is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

```
float array[];
```

The pair of brackets informs the compiler that the argument array is an array of numbers. It is not necessary to specify the size of the array here.

Example:

```
#include<stdio.h>
#include<conio.h>
float largest(array,size)
float array[];
int size;
 {
        int i;
        float max;
        max=array[0];
        for(i=1;i<size;i++)
          {
           if (max<array[i])
             max=array[i];
          }
        return(max);
 }
main()
 {
        int i,n;
        float a[20];
        printf("Enter the value of n : ");
        scanf("%d",&n);
        printf("Enter the values :\n");
        for(i=0;i<n;i++)
           scanf("%d",&a[i]);
        printf("Largest value is :%f",largest(a,n));
}
```

THE SCOPE AND LIFETIME OF VARIABLES IN FUNCTIONS

A variable in C can have any one of the four storage classes:

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

The scope of variable determines over what part(s) of the program a variable is actually available for use. Lifetime refers to the period during which a variable retains a given value during execution of a program (alive).

The variables may also be broadly categorized, depending on the place of their declaration, as internal (local) or external (global). Internal or local variables are those which are declared within a particular function, while external or global variables are declared outside of any function.

Automatic Variables

Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited. Automatic variables are also referred to as local or internal variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable.

Example:

```
#include<stdio.h>
#include<conio.h>
main()
 {
        auto int a,b,c;
        printf("Enter the values :");
        scanf("%d%d",&a,&b);
        c=a+b;
        printf("The sum is %d",c);
 }
```

Automatic variables can also be defined within a set of braces known as "blocks". They are meaningful only inside the blocks where they are defined. Consider the example below:

```
main()
  {
          int n,a,b;
          …..
          …..
          if (n<=100)
            {
                    int n,sum;
                    …..
                    …..
            }
  }
```

The variables **n, a** and **b** defined in main have scope from the beginning to the end of main. However, the variable n defined in the main cannot enter into the block of scope level 2 because the scope level 2 contains another variable named n. The second n is available only inside the scope level 2 and no longer available the moment control leaves the if block.

External Variables

Variables that are both alive and active throughout the entire program are known as external variables. They are also known as global variables. Global variables can be accessed by any function in the program. External variables are declared outside a function. For example, the external declaration of integer number and float length might appear as:

```
int no;
float len=9.5;
main()
  {
          …..
          …..
  }
fun1()
  {
          …..
          …..
  }
fun2()
  {
          …..
          …..
  }
```

The variables ***no*** and ***len*** are available for use in all the three functions.

Example:

```
#include<stdio.h>
int x;
main()
  {
        x=10;
        printf(“\nx=%d”,x);
        printf(“\nx=%d”,fun1());
        printf(“\nx=%d”,fun2());
        printf(“\nx=%d”,fun3());
  }
fun1()
  {
        x+=10;
        return(x);
  }
fun2()
  {
        int x;
        x=5;
        return(x);
  }
fun3()
  {
        x+=5;
        return(x);
  }
```

Output:

```
x=10
x=20
x=5
x=25
```

External Declaration

In the following program segment the main cannot access the variable y as it has been declared after the main function. This problem can be solved by declaring the variable with the storage class extern.

```
main()
  {
        y=5;
        …..
        …..
  }
int y;
fun1()
  {
        …..
        y++;
        …..
  }
```

For example:

```
main()
  {
        extern int y;    /* external declaration */
        …..
        …..
  }
fun1()
  {
        extern int y;    /* external declaration */
        …..
        …..
  }
int y;   /* definition */
```

Although the variable y has been defined after both the functions, the external declaration of y inside the function informs the compiler that y is an integer type defined somewhere else in the program.

Static Variables

The value of static variables persists until the end of the program. A variable can be declared static using the keyword static like

```
static int x;
static float y;
```

A static variable may be either an internal type or an external type, depending on the place of declaration.

Internal static variables are those which are declared inside a function. The scope of internal static variables extend upto the end of the function in which they are defined.

Example:

```
#include<stdio.h>
main()
  {
        int i;
        for(i=1;i<=10;i++)
          disp();
  }
disp()
  {
        static int x=0;
        x++;
        printf("x=%d",x);
  }
```

Register Variables

A variable can be kept in one of the machine's registers, instead of keeping in the memory. Since a register access is much faster than a memory access, keeping the frequently accessed variables (eg. Loop control variables) in the register will lead to faster execution of programs. This is done as follows:

register int count;

Example:

```
#include<stdio.h>
#include<conio.h>
main()
 {
        register int a,b,c;
        printf("Enter the values :");
        scanf("%d%d",&a,&b);
        c=a+b;
        printf("The sum is %d",c);
 }
```

The following table summarizes the information on the visibility and lifetime of variables in functions and files.

Storage Class	Where Declared	Visibility (Active)	Lifetime (Alive)
None	Before all functions in a file (may be initialized)	Entire file plus other fines where variable in declared with extern.	Entire program (Global)
extern	Before all functions in a file (cannot be initialized)	Entire file plus other files where variable is declared extern and the file where originally declared as global	Global
None or **auto**	Inside a function or block	Only in that function or block	Until end of function or block
register	Inside a function or block	Only in that function or block	Until end of function or block
static	Before all functions in a file	Only in that file	Global
static	Inside a function	Only in that function	Global

STRUCTURES AND UNIONS

Structure is a method for packing data of different types. A structure is a convenient tool for handling a group of logically related data items.

STRUCTURE DEFINITION

A structure definition creates a format that may be used to declare structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

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

The keyword struct declares a structure to hold the details of four fields, namely title, author, pages, and price. These fields are called structure elements or members. Each member may belong to a different type of data. book_bank is the name of the structure and is called the structure tag.

The general format of a structure definition is as follows:

```
struct tag_name
 {
   data_type member1;
   data_type member2;
       …..
       …..
 };
```

We can declare structure variables using the tag name. For example, the statement

```
struct book_bank b1,b2,b3;
```

declares b1,b2 and b3 as variables of type struct book_bank.

In defining a structure you may note the following syntax:

1. The template is terminated with a semicolon.
2. Each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name can be used to declare structure variables of its type.

It is also allowed to combine both the template declaration and variables declaration in one statement. The declaration

```
struct book_bank
 {
   char title[20];
   char author[15];
   int pages;
   float price;
 }b1,b2,b3;
```

is valid. The use of tag name is optional. For example,

```
struct
 {
   char title[20];
   char author[15];
   int pages;
   float price;
 }b1,b2,b3;
```

declares b1, b2, and b3 as structure variables representing three books.

GIVING VALUES TO MEMBERS

The link between a member and a variable is established using the member operator '.' Which is also known as 'dot operator' or 'period operator'. For example,
b1.price

Is the variable representing the price of b1 and can be treated like any other ordinary variable.

```
strcpy(b1.title,"BASIC");
strcpy(b1.author,"SAM");
b1.pages=255;
b1.price = 535.00;
```

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

```
scanf("%s",b1.title);
scanf("%s",b1.author);
scanf("%d",&b1.pages);
scanf("%d",&b1.price);
```

Example program:

```
#include<stdio.h>
#include<conio.h>
struct book_bank
 {
   char title[20];
   char author[15];
   int pages;
   float price;
 };
```

```
main()
 {
   struct book_bank b1;
   printf("Enter the book details\n");
   scanf("%s",b1.title);
   scanf("%s",b1.author);
   scanf("%d",b1.pages);
   scanf("%d",b1.price);
   printf("\nTitle : %s",b1.title);
   printf("\nAuthor : %s",b1.author);
   printf("\nPages : %d",&b1.pages);
   printf("\nPrice : %d",&b1.price);
   getch();
 }
```

STRUCTURE INITIALIZATION

A structure must be declared as static if it is to be initialized inside a function.

```
main()
 {
        static struct
         {
            int weight;
            float height;
         }s={57,175.70};
        …..
        …..
 }
```

This assigns the value 57 to s.weight and 175.70 to s.height.

The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main()
 {
        struct stu
         {
            int weight;
            float height;
         };
        static struct stu s1={57,175.70};
        static struct stu s2={60,179.50};
        …..
        …..
 }
```

Another method is to initialize a structure variable outside the function as shown below:

```
struct stu
 {
   int weight;
   float height;
 }s1={57,175.70};
main()
  {
        static struct stu s2={60,179.50};
        …..
        …..
  }
```

COMPARISON OF STRUCTURE VARIABLES

Two variables of the same structure type can be compared the same way as ordinary variables. If person1 and person2 belong to the same structure, then the following operations are valid:

Operation	Meaning
person1=person2	Assign person2 to person1
person1==person2	Compare all members of person1 and person2 and return1 if they are equal, 0 otherwise.
person1!=person2	Return 1 if all the members are not equal, 0 otherwise

Note that not all compilers support these operations. For example, Microsoft C version does not permit any logical operations on structure variables. In such case, individual members can compared using logical operators.

Example:

```
#include<stdio.h>
struct stu
 {
   int weight;
   float height;
 };
main()
  {
        struct stu s1,s2;
        printf("Enter s1.weight : ");
        scanf("%d",&s1.weight);
        printf("Enter s1.height : ");
        scanf("%f",&s1.height);
        printf("Enter s2.weight : ");
        scanf("%d",&s2.weight);
        printf("Enter s2.height : ");
        scanf("%f",&s2.height);
        if (s1==s2)
          printf("s1==s2");
        else
          printf("s1!=s2");
    }
```

ARRAY OF STRUCTURES

In array of structures, each element of the array representing a structure variable. For example,

struct student s[50];

defines an array called s, that consists of 50 elements. Each element is defined to be of the type struct student. Consider the following declaration:

```
struct marks
  {
        int m1;
        int m2;
        int m3;
  };

main()
  {
        static struct marks s[2]={{89,87,99},{76,98,90}};
  }
```

This declares the s as an array of three elements s[0] and s[1] and initializes their members as follows:

```
s[0].m1=89;
s[0].m2=87;
s[0].m3=99;
s[1].m1=76;
s[1].m2=98;
s[1].m3=90;
```

The array s actually looks as shown in the following fig.

s[0].m1	**89**
s[0].m2	**87**
s[0].m3	**99**
s[1].m1	**76**
s[1].m2	**98**
s[1].m3	**90**

Example Program:

```
#include<stdio.h>
struct student
 {
   int rno;
   char name[20];
   int m[5];
   int tot;
   float avg;
};

main()
 {
   struct student s[50];
   int n,i,j,f;
   printf("Enter total no. of students : ");
   scanf("%d",&n);
   for(i=0;i<n;i++)
    {
         printf("Enter Roll No. : "); scanf("%d",&s[i].rno);
         printf("Enter Name. : "); scanf("%s",s[i].name);
         s[i].tot=0;
         for(j=0;j<5;j++)
          {
           printf("Enter m%d : ",j+1); scanf("%d",&s[i].m[j]);
           s[i].tot+=s[i].m[j];
          }
         s[i].avg=s[i].tot/5.0;
         for(i=0;i<n;i++)
         {
           printf("\nRoll No. : %d",s[i].rno);
           printf("\nName. : %s",s[i].name);
           for(j=0;j<5;j++)
            {
             printf("\nM%d : %d",j+1,s[i].m[j]);
            }
          printf("\nTotal : %d",s[i].tot);
          printf("\nAverage : %f",s[i].avg);
         }
}
```

ARRAYS WITHIN STRUCTURES

C permits the use of arrays as structure members. For example, the following structure declaration is valid:

```
struct marks
  {
        int rno;
        int m[3];
  }s[2];
```

Here, the member m contains three elements, m[0], m[1] and m[2]. These elements can be accessed using appropriate subscripts. For example, the name

s[0].m[0]

would refer to the marks obtained in the first subject by the first student.

Example Program:

```
#include<stdio.h>
struct student
 {
  int rno;
  char name[20];
  int m[5];
  int tot;
  float avg;
};

main()
 {
  struct student s[50];
  int n,i,j,f;
  printf(“Enter total no. of students : “);
  scanf(“%d”,&n);
  for(i=0;i<n;i++)
   {
        printf(“Enter Roll No. : “); scanf(“%d”,&s[i].rno);
        printf(“Enter Name. : “); scanf(“%s”,s[i].name);
        s[i].tot=0;
        for(j=0;j<5;j++)
         {
           printf(“Enter m%d : “,j+1); scanf(“%d”,&s[i].m[j]);
           s[i].tot+=s[i].m[j];
          }
        s[i].avg=s[i].tot/5.0;
        for(i=0;i<n;i++)
        {
          printf(“\nRoll No. : %d“,s[i].rno);
          printf(“\nName. : %s“,s[i].name);
          for(j=0;j<5;j++)
           {
            printf(“\nM%d : %d“,j+1,s[i].m[j]);
           }
         printf(“\nTotal : %d“,s[i].tot);
         printf(“\nAverage : %f“,s[i].avg);
        }
 }
```

STRUCTURES WITHIN STRUCTURES

Structures within a structure means nesting of structures. Nesting of structures is permitted in C.

Example:

```
struct student
 {
  int rno;
  char name[20];
  struct
   {
      int dd;
      int mm;
      int yy;
   }dob;
  int m[5];
  int tot;
  float avg;
}s;
```

The student structure contains a member named dob which itself is a structure with three members. The members contained in the inner structure namely dd, mm, and yy can be referred to as

```
s.dob.dd
s.dob.mm
s.dob.yy
```

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct student
 {
  int rno;
  char name[20];
  struct
   {
      int dd;
      int mm;
      int yy;
   }dob,doj;
  int m[5];
  int tot;
  float avg;
}s;
```

The inner structure has two variables, dob and doj. This implies that both of them have the same structure template. A base member can be accessed as follows:

```
s.dob.dd
s.doj.dd
```

We can also use tag names to define inner structures. Example:

```
struct date
 {
   int dd;
   int mm;
   int yy;
 };
struct student
 {
  int rno;
  char name[20];
  struct date dob;
  struct date doj;
  int m[5];
  int tot;
  float avg;
}s;
```

date template is defined outside the salary template and is used to define the structure of dob and doj inside the student structure.

STRUCTURE AND FUNCTIONS

C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from on function to another.

1. The first method is to pass each member of the structure as an actual argument of the function call.
2. The second method involves passing of a copy of the entire structure to the called function.
3. The third approach employs a concept called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the called function.

The general format of sending a copy of a structure to the called function is:

```
function name(structure variable name)
```

The called function takes the following form:

```
data_type function name(st_name)
struct_type st_name;
 {
        …..
        …..
        return(expression);
 };
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same struct type.
3. The return statement is necessary only when the function is returning some data. The expression may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called function must be declared in the calling function for its type, if it is placed after the falling function.

Example Program:

```
#include<stdio.h>
struct student
 {
   int rno;
   char name[20];
   int m[5];
   int tot;
   float avg;
 };

void display(stu)
struct student stu;
 {
  int j;
  printf("\nRoll No. : %d",stu.rno);
  printf("\nName. : %s",stu.name);
  for(j=0;j<5;j++)
   {
       printf("\nM%d : %d",j+1,stu.m[j]);
   }
  printf("\nTotal : %d",stu.tot);
  printf("\nAverage : %f",stu.avg);
 }

main()
 {
   struct student s[50];
   int n,i,j,f;
   printf("Enter total no. of students : ");
   scanf("%d",&n);
   for(i=0;i<n;i++)
    {
         printf("Enter Roll No. : "); scanf("%d",&s[i].rno);
         printf("Enter Name. : "); scanf("%s",s[i].name);
```

```
                    s[i].tot=0;
                    for(j=0;j<5;j++)
                     {
                       printf("Enter m%d : ",j+1); scanf("%d",&s[i].m[j]);
                       s[i].tot+=s[i].m[j];
                      }
                    s[i].avg=s[i].tot/5.0;
                    for(i=0;i<n;i++)
                    {
                      display(s[i]);
                    }
            }
```

UNIONS

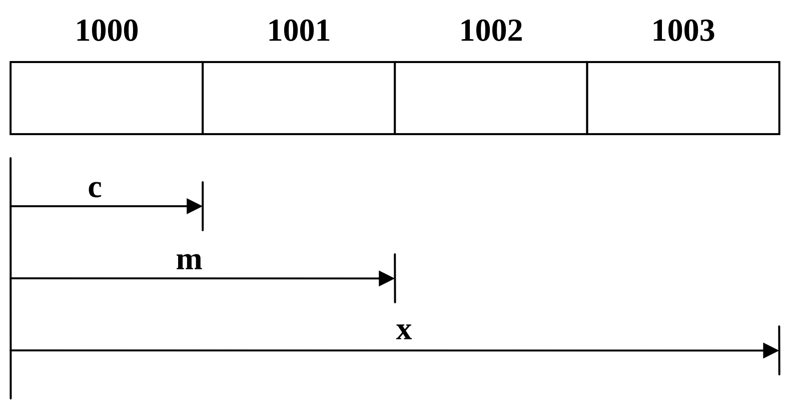
Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. Like structures, a union can be declared using the keyword union as follows:

```
union item
  {
        int m;
        float x;
        char c;
  }code;
```

This declares a variable code of type union item. The union contains three members, each with a different data type. However, we can use only one of them at a time.

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members. The following Fig. shows how all the three variables share the same address.

Storage of 4 bytes



We can use the same syntax that we use for structure members. That is,

```
code.m
code.x
code.c
```

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statements such as

```
code.m=789;
code.x=34.234;
printf("%d",code.m);   /* erroneous output */
```

would produce erroneous output.

SIZE OF STRUCTURES

The actual size of structure variables in terms of bytes may change from machine to machine. We may use the unary operator sizeof to tell us the size of a structure. The expression

```
sizeof(struct x);
```

will evaluate the number of bytes required to hold all the members of the structure x. The sizeof operator would be useful to determine the number of records in a database.

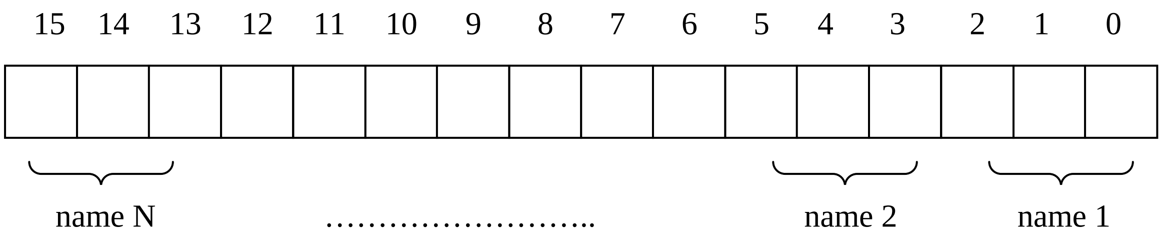
BIT FIELDS

C permits us to use small bit fields to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits as if it represented an integral quantity.

A bit field is a set of adjacent bits whose size can be from 1 to 16 bits in length. A word can therefore be divided into a number of bit fields. The name and size of bit fields are defined using a structure. The general form of bit field definition is:

```
struct tag_name
  {
        data-type name1:bit-length;
        data-type name2:bit-length;
        data-type name3:bit-length;
                .....
                .....
        data-type namen:bit-length;
  };
```

The data-type is either int or unsigned int or signed int and the bit-length is the number of bits used for the specified name. The field name is followed by a colon. The bit-length is decided by the range of value to be stored. The largest value that can be stored is 2^n-1, where n is bit-length. The sketch below illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.



There are several specific points to observe:

1. The first field always starts with the first bit of the world.
2. A bit field cannot overlap integer boundaries.
3. There can be unnamed fields declared with size,
   Example: unsigned : bit-length;
4. There can be unused bits in a word.
5. We cannot take the address of a bit field variable.
6. Bit fields cannot be arrayed.
7. Bit fields should be assigned values that are within the range of their size.

Suppose, we want to store and sue personal information of employees in compressed form. This can be done as follows:

```
struct personal
 {
       unsigned sex  :        1
       unsigned age  :        7
       unsigned m-status:     1
       unsigned children:     3
       unsigned :             4
 }emp;
```

This defines a variable name emp with four fields. The range of values each field could have is as follows:

Bit-field	Bit-length	Range of values
sex	1	0 or 1
age	7	0 or 127 (2^7-1)
m_status	1	0 or 1
children	3	0 to 7 (2^3-1)

MORE ON FUNCTIONS

PREPROCESSOR

The preprocessor is a program that modifies the C source program according to directives supplied in the program. The preprocessor does not modify the program file, but creates a new file that contains the processed version of the program. This new file is then submitted to the compiler. A preprocessor carries out the following actions on the source file.

1. Replacement of defined identifiers by pieces of text
2. Conditional selection of parts of the source file.
3. Inclusion of other files
4. Renumbering of source files and the renaming of the source files itself.

The general rules for defining a preprocessor are

1. All preprocessor directives begin with the sharp sign(#).
2. They must start in the first column.
3. The preprocessor directive is terminated not by a semicolon.
4. Only one preprocessor directive can occur on a line.
5. The preprocessor directives may appear at any place in the source file.

The C preprocessor is a simple macro processor. The preprocessor is controlled by special preprocessor command lines. The common C preprocessor directives and their uses are

Directive	Uses
# include	Insert text from another file
#define	Define preprocessor macro
#undef	Remove macro definitions
#if	Conditionally include some text, based on the value of the constant expression
#ifdef	Conditionally include some text, based on whether a macro name is defined
#ifndef	Conditionally include some text, with the sense of the test opposite that of #ifdef
#else	Alternatively include some text, if the previous #if, #ifdef, or #ifndef test failed
#elif	Combination of #if and #else
#endif	Terminate conditional text
#line	Give a line number for compiler messages
#error	Terminate processing early

MACROS

The single line abbreviation for a group of lines called macro. The macros can be classified into two groups. They are

1. Simple macro definitions
2. Macro definition with arguments.

The **advantages** of using macro in C are

1. Easy to read and write
2. Easy to check the string constants and debug
3. Easy to transfer from one machine to other
4. Gives good look to the program.

<u>Simple macro definitions</u>

A macro is simply a substitution string that is placed in the program. For an example,

```
#define MAX 100
main()
 {
   char name[MAX];
        for( i= 0;i<=MAX – 1; i++)
        ………………
        ………………
 }
```

which is internally replaced with the following program

```
main()
 {
   char name[100];
        for( i= 0;i<=100 – 1; i++)
        ………………
        ………………
 }
```

Each occurrence of the identifier MAX as a token is replaced with the string 100.

The simple form of macro is particularly useful for introducing named constant into a program.
For example

```
        #define TRUE 1
        #define FALSE 0
        #define EOF -1
        #define SIZE 5
        #define MAX 100
```

<u>Macro with parameters</u>

The more complex form of macro definition declares the names of formal parameters within parentheses, separated by commas. The general form is

```
#define name(variable1, variable2, variable3, …….. , variable n) substitution_string
```

For example:

```
        #define PRODUCT (x,y) ((x)*(y))
        #define SUM (x,y) ((x)+(y))
        #define MAX (x,y) ((x)>(y)?(x):(y))
```

Macros operate purely by textual substitution of tokens.

For example,

```
#define PRODUCT (x)         x*x
```

if this macro definition is called in the C program

```
#define PRODUCT (x)         x*x
main()
 {
        int a;
        a=PRODUCT(10);
        printf("%d",a);
 }
```

The above program is internally replaced by

```
main()
 {
        int a;
        a=10*10;
        printf("%d",a);
 }
```

Other preprocessing Techniques

Since the preprocessor merely substitutes a string of text for another without doing any checking, a wide variety of substitutions is possible, even making a typed source program look like another language. Many Pascal symbols can be used in C programs by just including many # define statements to convert them to legal C symbols.

For example,

```
/* pascal.h */
#define begin         {
#define end           }
#define writeln       printf
#define program       main
#define integer       int
```

A program to define the Pascal syntax in the C program using a macro declaration is given below

```
# include<stdio.h>
# include<pascal.h>
program()
begin
        writeln("Pascal Output");
end
```

Conditional compilation

The preprocessor conditional compilation commands allow lines of source text to be parsed through or eliminated by the preprocessor on the basis of a computed condition. The preprocessor conditional commands are

```
#if
#else
#endif
#elif
```

The general form is

```
#if constant expression
        group of lines 1
#else
        group of lines 2
#endif
```

A group of lines may contain any number of lines. The #else command may be omitted.

The #elif command

The #elif command used between #if and #endif in the same way on #else but has a constant expression to evaluate in the same way as #if.
The general form is

```
#if constant expression
        group of lines 1
#elif constant expression
        group of lines 2
#elif constant expression
        group of lines 3
……………..
……………..
#else
        group of lines n
#endif
```

For example,

```
#if  expression
        ……………..
        ……………..
#elif  expression
        ……………..
        ……………..
#elif  expression
        ……………..
        ……………..
#else
        ……………..
        ……………..
#endif
```

HEADER FILES

A header file contains the definition, global variable declarations, and initialization by all the file in a program. The header file can be included in the C program using the macro definition “#include” command.

For example,

```
# include<stdio.h>
#include<conio.h>
        or
# include"stdio.h"
#include"conio.h"
```

Some of the header files used in C are

stdio.h
conio.h
time.h
math.h
string.h
dir.h
dos.h
process.h
ctype.h
io.h

STANDARD FUNCTIONS

The standard libraries are used to perform some predefined operations on characters, strings etc. The standard libraries are also called library functions or built in functions or predefined functions. Most of the C compilers support the following standard library facilities.

- operations on characters
- operations on strings
- mathematical operations
- storage allocations procedures
- input/output operations

math.h

The mathematical library functions can be defined in one of the following ways

```
# include <math.h>
        or
# include "math.h"
```

abs()

The abs() function is used to find the absolute value. The data type of both argument and the result are int.

Example :

```
Abs(100)      =     100
Abs(-100)     =     100

# include<stdio.h>
# include<math.h>
main()
 {
   printf("Absolute value of abs(-100) is %d",abs(-100));
 }
```

The result is

```
Absolute value of abs(-100) is 100
```

<u>fabs()</u>

The fabs() function is used to find the absolute value of the floating point number. The data type of both argument and the result are double.

Example :

```
fabs(100.80)    =     100.80
fabs(-100.80)   =     100.80

# include<stdio.h>
# include<math.h>
main()
 {
   printf("Absolute value of fabs(-100.80) is %d",fabs(-100.80));
 }
```

The result is

```
Absolute value of fabs(-100.80) is 100.80
```

<u>acos()</u>

The acos() function is used to find arc cosine value. The data type of both argument and the result are double. The result is in radians and lies between 0 to phi.

Example :

```
acos(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("acos(x) is %f",acos(x));
 }
```

asin()

The asin() function is used to find arc sine value. The data type of both argument and the result are double. The result is in radians and lies between –(phi/2) to (phi/2).

Example :

```
asin(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("asin(x) is %f",asin(x));
 }
```

atan()

The atan() function is used to find arc cosine value. The data type of both argument and the result are double. The result is in radians and lies between –(phi/2) to (phi/2).

Example :

```
atan(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("atan(x) is %f",atan(x));
 }
```

ceil()

The ceil() function is used to round its argument upto an integer. The data type of both argument and the result are double.

Example:

```
ceil(4.9)       =       5

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("ceil(x) value %d",ceil(x));
 }
```

floor()

The floor() function is used to round its argument down to an integer. The data type of both argument and the result are double.

Example:

```
floor(4.4)      =       4

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("floor(x) value %d",floor(x));
 }
```

cos()

The cos() function is used to find trigonometric cosine value. The argument is taken to be in radians. The data type of both argument and the result are double.

Example :

```
cos(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("cos(x) is %f",cos(x));
 }
```

sin()

The sin() function is used to find trigonometric sine value. The argument is taken to be in radians. The data type of both argument and the result are double.

Example :

```
sin(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("sin(x) is %f",sin(x));
 }
```

tan()

The tan() function is used to find trigonometric tangent value. The argument is taken to be in radians. The data type of both argument and the result are double.

Example :

```
tan(x)

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

main()
 {
    double x;
    printf("tan(x) is %f",tan(x));
 }
```

cosh()

The cosh() function is used to find hyperbolic cosine value. The argument is taken to be in radians. The data type of both argument and the result are double.

Example :

```
cosh(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("cosh(x) is %f",cosh(x));
 }
```

sinh()

The sinh() function is used to find hyperbolic sine value. The argument is taken to be in radians. The data type of both argument and the result are double.

Example :

```
sinh(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("sinh(x) is %f",sinh(x));
 }
```

tanh()

The tanh() function is used to find hyperbolic tangent value. The argument is taken to be in radians. The data type of both argument and the result are double.

Example :

```
tanh(x)
```

```
# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("tanh(x) is %f",tanh(x));
 }
```

<u>exp()</u>

The exp() function is used to find exponential value, ie. e raised to the power x. The data type of both argument and the result are double.

Example :

```
exp(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("exp(x) is %f",exp(x));
 }
```

<u>log()</u>

The log() function is used to find natural logarithms. The data type of both argument and the result are double.

Example:

```
log(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("log(x) value %f",log(x));
 }
```

<u>log10()</u>

The log10() function is used to find base 10 logarithms. The data type of both argument and the result are double.

Example:

```
Log10(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("log10(x) value %f",log10(x));
 }
```

pow()

The log() function is used to find x raised to the power y. The data type of both argument and the result are double.

Example:

```
pow(x,y)

# include<stdio.h>
# include<math.h>
main()
 {
    double x,y;
    printf("pow(x,y) value %f",pow(x,y));
 }
```

sqrt()

The sqrt() function is used to find square root of the arguments. The data type of both argument and the result are double.

Example:

```
sqrt(x)

# include<stdio.h>
# include<math.h>
main()
 {
    double x;
    printf("sqrt(x) value %f",sqrt(x));
 }
```