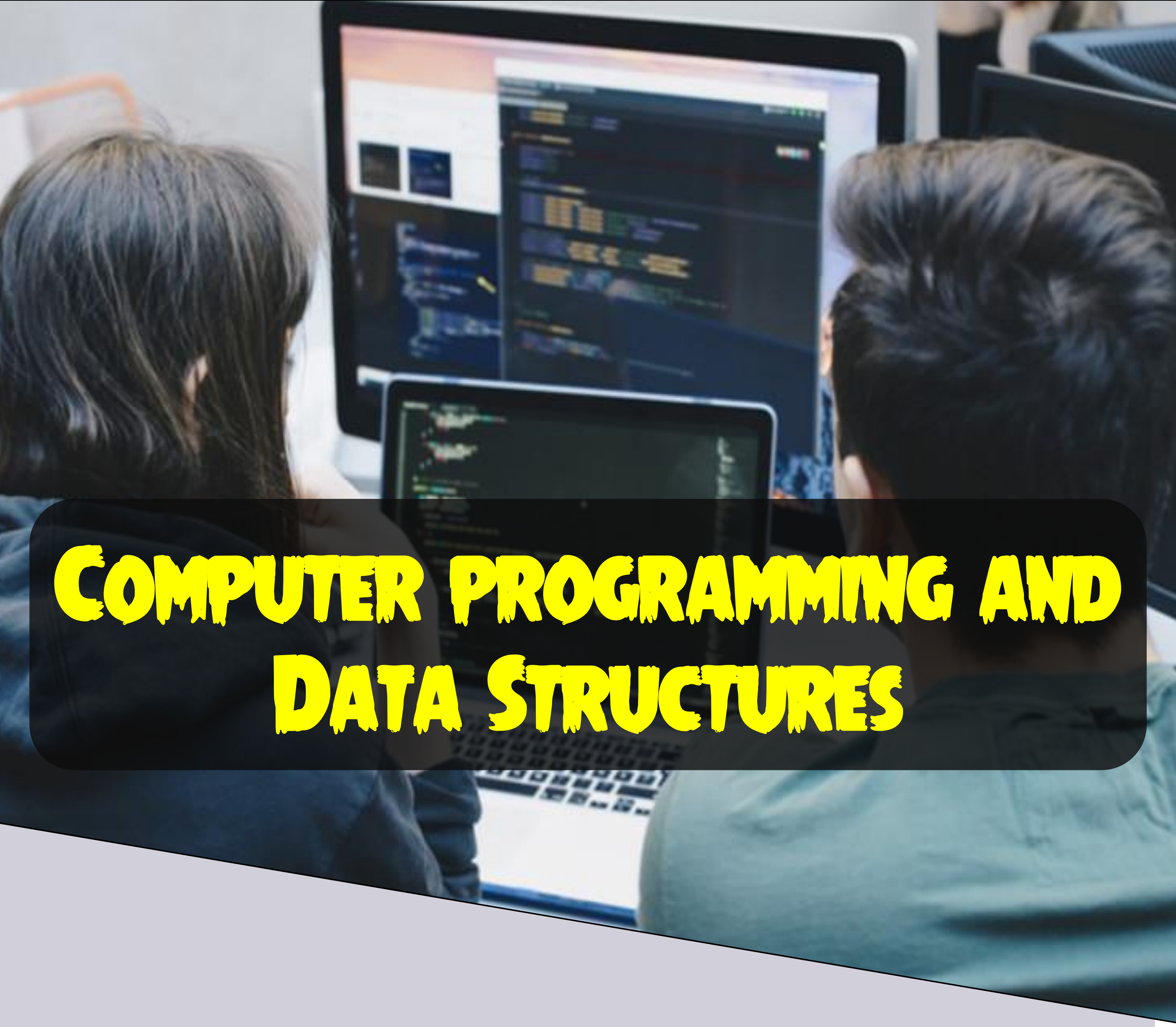


Computer programming and Data Structures

P.S. Parsania

Computer programming and Data Structures

Author

P.S. Parsania

AAU, Anand

INDEX

Module - 1 C Language Fundamentals

Lesson-1 Introduction to C Language

1.1 INTRODUCTION

C is a powerful, flexible, portable and structured programming language. Since it allows you to develop programs using well-defined control structures and provides modularity (breaking the task into multiple sub tasks that are simple enough to understand and to reuse). C is often called a middle-level language because it combines the best elements of low-level or machine language with high-level languages. Middle-level implies that it is neither high level (i.e. using easy to use English words and structures that are hard to understand by the machine but easy to understand by the programmer) nor low level (i.e. using obscure assembly language that describes how the computer does its processing at the hardware [chip] level, but works very fast due to it being written using a language the machine understands more readily than humans writing programs).

1.2 HISTORY OF C LANGUAGE

C was developed by Dennis Ritchie at Bell Laboratories in 1972. Most of its principles and ideas were taken from the earlier language B, BCPL and CPL. CPL was developed jointly between the Mathematical Laboratory at the University of Cambridge and the University of London Computer Unit in 1960s. CPL (Combined Programming Language) was developed with the purpose of creating a language that was capable of both machine independent programming and would allow the programmer to control the behavior of individual bits of information. But the CPL was too large for use in many applications. In 1967, BCPL (Basic Combined Programming Language) was created as a scaled down version of CPL while still retaining its basic features. This process was continued by Ken Thompson. He made B Language during working at Bell Labs. B Language was a scaled down version of BCPL. B Language was written for the systems programming. In 1972, a co-worker of Ken Thompson, Dennis Ritchie developed C Language by taking some of the generality found in BCPL to the B language.

The original PDP-11 version of the Unix system was developed in assembly language. In 1973, C language had become powerful enough that most of the Unix kernel was rewritten in C. This was one of the first operating system kernels implemented in a language other than assembly.

During the rest of the 1970's, C spread throughout many colleges and universities because of its close ties to UNIX and the availability of C compilers. Soon, many different organizations began using their own versions of C Language. This was causing great compatibility problems. In 1983, the American National Standards Institute (ANSI) formed a committee to establish a standard definition of C Language. That is known as ANSI Standard C. Today C is the most widely used System Programming Language.

1.3 LEVEL OF PROGRAMMING LANGUAGES

There are three different levels of programming languages. They are

- Machine languages (low level)
- Assembly languages
- Procedure Oriented languages (high level)

1.3.1 Machine Languages:-

Computers are made up of number of electronic components and they all are two – state electronic components means they understand only the 0 – (pulse) and 1 (non – pulse). Therefore the instruction given to the computer must be written using binary numbers i.e. 1 and 0. Computers do not understand English or any other language but they only understand or respond to binary numbers (0 and 1). So each computer has its own Machine languages.

1.3.2 Assembly Languages:-

As it was very tedious to understand and remember 0's representing numerous data and instruction. So to resolve this problem mnemonics codes were developed. For example add is used as symbolic code to represent addition. SUB is used for subtraction. They are the symbolic representation of certain combination of binary numbers for example **S U B** Which represents certain actions as computer understand only Machine language instructions a program written in Assembly Language must be translated into Machine languages. Before it can be executed this translation if done by another program called assembler. This assembler will translate mnemonic codes into Machine languages.

1.3.3 Procedure Oriented Languages

In earlier assembly languages assembler programs produced only one Machine languages instruction for every assembly languages instruction. So to resolve this problem now assembler was introduced. It can produce several machine level instructions for one assembly language instruction. Thus programmer was relieved from the task of writing and instruction for every machine operation performed. These languages contain set of words and symbols and one can write program with the combination of these keywords. These languages are also called high level languages. Basic, FORTRAN, Pascal and COBOL. The most important characteristic of high level languages is that it is machine independent and a program written in high level languages can be run on any computers with different architecture with no modification or very little modification.

1.4 LANGUAGE TRANSLATORS

As we know that computer understands only the instruction written in the Machine language. Therefore a program written in any other language should be translated to Machine language. For these purpose special programs are available they are called translators or language processors. These special programs accept the user program and check each statement and produce a corresponding set of Machine language instructions. There are two types of Translators.

1.4.1 Compiler:-

A Compiler checks the entire program written by user and if it is free from error and mistakes then produces a complete program in Machine language known as object program. But if it founds some error in program then it does not execute the single statement of the program. So compiler translate whole program in Machine language before it starts execution.

1.4.2 Interpreters:-

Interpreters perform the similar job like compiler but in different way. It translates (Interprets) one statement at a time and if it is error – free then executes that statement. This continues till the last statement in the program has been translated and executed. Thus Interpreter translates and executes the statement before it goes to next statement. When it founds some error in statement it will immediately stop the execution of the program. Since compiler of Interpreter can translate only a particular language for which it is designed one has to use different compiler for different languages.

1.5 Difference between Compiler – Interpreter:-

Error finding is much easier in Interpreter because it checks and executes each statement at a time. So wherever it finds some error it will stop the execution. Where Compiler first checks all the statement for error and provide lists of all the errors in the program. Interpreter take more time for the execution of a program compared to Compilers because it translates and executes each statement one by one.

1.6 FEATURES OF C LANGUAGE

There are several reasons why many computer professionals feel that C is at the top of the list:

1.6.1 FLEXIBILITY

C is a powerful and flexible language. C is used for projects as diverse as operating system, word processor, graphics, spreadsheets and even compilers for other language. C is a popular language preferred by professional programmers. As a result, a wide variety of C compilers and helpful accessories are available.

1.6.2 PORTABILITY

C is a portable language. Portable means that a C program written for one computer system can be run on another system with little or no modification. Portability is enhanced by the ANSI standard by C, the set of rules for C compilers.

1.6.3 COMPACTNESS

C is a language of few words, containing only a handful terms, called keywords, which serve as the base on which the language's functionality is built. You might think that a language with more keyword would be more powerful. This isn't true. As you will find that it can be programmed to do any task.

1.6.4 REUSABILITY:

C is modular, C code can and should be written in routine called functions. These functions can be reused in other applications or programs. By passing pieces of information to the function, you can create useful, reusable code.

1.7 BASIC STRUCTURE OF C PROGRAM

C program can be viewed as group of building blocks called functions. A function is a subroutine that may include one or more statement designed to perform a specific task. C program may contain one or more section as shown below:

Documentation Section
Link Section
Definition Section
Global declaration Section

Main() function section
{
Declaration Part
Executable Part
}

Subprogram Section
Function1 ()
{
 	
}
Function2 ()
{
 	
}

1.7.1 Documentation Section:

This section contains set of comments lines consist of details like program name, author name and purpose or functionality of the program.

1.7.2 Link Section:

This section consists of instructions to be given to the compiler to link functions from the system library. For example if you want to use some mathematical function then you have to define link for math.h file in link section. For Example

```
# include<stdio.h>

# include<math.h>
```

1.7.3 Definition Section:

This section defines all the symbolic constants. For example PI=3.14. By defining symbolic constant one can use these symbolic constant instead of constant value.

```
# define PI 3.14

# define Temp 35
```

1.7.4 Global Declaration Section:

This section contains the declaration of variables which are used by more than one function of the program.

1.7.5 Main Function Section:

A main() function is a heart of any ‘C’ language program. Any C program is made up of 1 or more then 1 function. If there is only 1 function in the program then it must be the main program. An execution of any C program starts with main() function.

1.7.6 Subprogram or Sub Function Section:

They are the code sections which are define outside the boundary of main function. This function can be called from any point or anywhere from the program. Generally they are defining to solve some frequent tasks.

1.8 An Example of C Program

```
/* This program prints a one-line message */

#include <stdio.h>

void main()

{

printf("Hello World\n");

}
```

/* This program ... */	The symbols /* and */ delimit a comment. Comments are ignored by the compiler, and are used to provide useful information for *humans* that will read the program.
void	void keyword indicates that the main function is not going to return any value. More about function and return type in chapter 8. User defined function.
main()	C programs consist of one or more functions. One and *only* one of these functions must be called main. The brackets following the word main indicate that it is a function and not a variable.
{}	braces surround the body of the function, which consists of one or more instructions (*statements*).
printf()	is a library function that is used to print on the standard output stream usually the screen.
"Hello World\n"	is a string constant.
\n	is the newline character.
;	a semicolon terminates a statement.

C is case sensitive, so the names of the functions (main and printf) must be typed in lower case as above.

With a few exceptions, any amount of white space (spaces, tabs and newlines) can be used to make your programs more readable. There are many conventions for program layout, just choose one that suits you, or alternatively get a program to format your code for you (such as the indent program).

Lesson-2 Constant, Variables and Data Types

2.1 INTRODUCTION

As every natural language has a basic character set, computer languages also have a character set, rules to define words. Words are used to form statements. These in turn are used to write the programs.

Computer programs usually work with different types of data and need a way to store the values being used. These values can be numbers or characters. C language has two ways of storing number values—**variables and constants**—with many options for each. Constants and variables are the fundamental elements of each program. Simply speaking, a program is nothing else than defining them and manipulating them. A variable is a data storage location that has a value that can change during program execution. In contrast, a constant has a fixed value that can't change.

This unit is concerned with the basic elements used to construct simple C program statements. These elements include the C character set, identifiers and keywords, data types, constants, variables and arrays, declaration and naming conventions of variables.

2.2 CHARACTER SET

When you write a program, you express C source files as text lines containing characters from the character set. When a program executes in the target environment, it uses characters from the character set. These character sets are related, but need not have the same encoding or all the same members.

Every character set contains a distinct code value for each character in the basic C character set. A character set can also contain additional characters with other code values. The C language character set has alphabets, numbers, and special characters as shown below:

- Alphabets including both lowercase and uppercase alphabets - A-Z and a-z.
- Numbers 0-9
- Special characters include:

;	:	{	,	‘	“	|
}	>	<	/	\	~	_
[	]	!	$	?	*	+
=	(	)	-	%	#	^
@	&	.				

2.3 IDENTIFIERS AND KEYWORDS

Identifiers are the names given to various program elements such as constants, variables, function names and arrays etc. Every element in the program has its own distinct name but one cannot select any name unless it conforms to valid name in C language. Let us study first, the rules to define names or identifiers.

2.3.1 Rules for Forming Identifiers

Identifiers are defined according to the following rules:

1. It consists of letters and digits.

2. First character must be an alphabet or underscore.

3. Both upper and lower cases are allowed. Same text of different case is not equivalent,

for example: **TEXT** is not same as **text**.

4. Except the special character underscore (_), no other special symbols can be used.

For example, some valid identifiers are shown below:

X

X123

_XI

temp

tax_rate

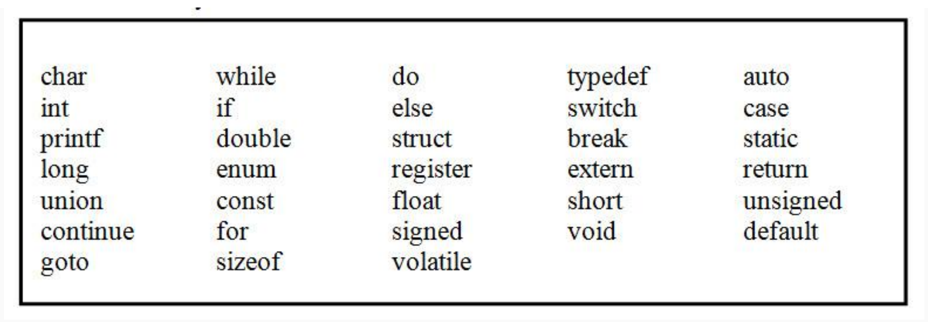
For example, some invalid identifiers are shown below:

123	First character to be alphabet.
"X."	Not allowed.
order-no	Hyphen not allowed.
error flag	Blank space not allowed.

2.3.2 Keywords

Keywords are reserved words which have standard, predefined meaning in C. They cannot be used as program-defined identifiers.

The lists of C keywords are as follows:



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

Note: Generally all keywords are in lower case although uppercase of same names can be used as identifiers.

2.4 DATA TYPES AND STORAGE

To store data inside the computer we need to first identify the type of data elements we need in our program. There are several different types of data, which may be represented differently within the computer memory. The data type specifies two things:

1. Permissible range of values that it can store.

2. Memory requirement to store a data type.

C Language provides four basic data types viz. int, char, float and double. Using these, we can store data in simple ways as single elements or we can group them together and use different ways (to be discussed later) to store them as per requirement. The four basic data types are described in the following table 2.1

Table 2.1: Basic Data Types

DATA TYPE	TYPE OF DATA	MEMORY	RANGE
int	Integer	2 Bytes	− 32,768 to 32,767
char	character	1 Byte	− 128 to 128
float	Floating point number	4 bytes	3.4e − 38 to 3.4e +38
double	Floating point number with higher precision	8 bytes	1.7e − 308 to 1.7e + 308

Memory requirements or size of data associated with a data type indicates the range of numbers that can be stored in the data item of that type.

2.5 DATA TYPE QUALIFIERS

Short, long, signed, unsigned are called the data type qualifiers and can be used with any data type. A short int requires less space than int and long int may require more space than int. If int and short int takes 2 bytes, then long int takes 4 bytes.

Unsigned bits use all bits for magnitude; therefore, this type of number can be larger. For example signed int ranges from –32768 to +32767 and unsigned int ranges from 0 to 65,535. Similarly, char data type of data is used to store a character. It requires 1 byte. Signed char values range from –128 to 127 and unsigned char value range from 0 to 255. These can be summarized as follows:

Table 2.2: Data Types qualifiers

Data type	Size (bytes)	Range
Short int or int	2	−32768 to 32767
Long int	4	−2147483648 to 2147483647
Signed int	2	−32768 to 32767
Unsigned int	2	0 to 65535
Signed char	1	−128 to 127
Unsigned char	1	0 to 255

2.6 VARIABLES

Variable is an identifier whose value changes from time to time during execution. It is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there. A variable represents a single data item i.e. a numeric quantity or a character constant or a string constant. Note that a value must be assigned to the variables at some point of time in the program which is termed as assignment statement. The variable can then be accessed later in the program. If the variable is accessed before it is assigned a value, it may give garbage value. The data type of a variable doesn't change whereas the value assigned to can change. All variables have three essential attributes:

• the name

• the value

• the memory, where the value is stored.

For example, in the following C program **a, b, c, d** are the variables but variable **e** is not declared and is used before declaration. After compiling the source code and look what gives?

```
void main( )
{
        int a, b, c;
        char d;
        a = 3;
        b = 5;
        c = a + b;
        d = 'a';
           e=d;
        ……….
        ……….
}
```

After compiling the code, this will generate the message that variable **e** not defined.

2.7 DECLARING VARIABLES

Before any data can be stored in the memory, we must assign a name to these locations of memory. For this we make declarations. Declaration associates a group of identifiers with a specific data type. All of them need to be declared before they appear in program statements, else accessing the variables results in junk values or a diagnostic error. The syntax for declaring variables is as follows:

data- type variable-name(s);

For example,

```
    int a;
  short int a, b;
  int c, d;
  long c, f;
  float r1, r2;
```

2.8 INITIALISING VARIABLES

When variables are declared initial, values can be assigned to them in two ways:

- Within a Type declaration

The value is assigned at the declaration time.

For example,

int a = 10;

float b = 0.4 e –5;

char c = ‘a’;

b) Using Assignment statement

- The values are assigned just after the declarations are made.

For example,

a = 10;

b = 0.4 e –5;

c = ‘a’;

2.9 CONSTANTS

A constant is an identifier whose value cannot be changed throughout the execution of a program whereas the variable value keeps on changing. In C there are four basic types of **constants**. They are:

1. Integer constants

2. Floating point constants

3. Character constants

4. String constants

Integer and Floating Point constants are numeric constants and represent numbers.

Rules to form Integer and Floating Point Constants

- No comma or blank space is allowed in a constant.
- It can be preceded by – (minus) sign if desired.
- The value should lie within a minimum and maximum permissible range decided by the word size of the computer.

2.9.1 Integer Constants

Further, these constant can be classified according to the base of the numbers as:

1. **Decimal integer constants**

These consist of digits 0 through 9 and first digit should not be 0.

For example,

1 443 32767

are valid decimal integer constants.

2. **Invalid Decimal integer Constants**

12 ,45	, not allowed
36.0	Illegal char.
1 010	Blank space not allowed
10 – 10	Illegal char –
0900	The first digit should not be a zero

3. **Octal integer constants**

These consist of digits 0 through 7. The first digit must be zero in order to identify the constant as an octal number.

Valid Octal INTEGER constants are:

0 01 0743 0777

Invalid Octal integer constants are:

743	does not begin with 0
0438	illegal character 8
0777.77	illegal char .

4. **Hexadecimal integer constants** 42 **Variables and Constants**

These constants begin with 0x or OX and are followed by combination of digits taken from hexadecimal digits 0 to 9, a to f or A to F.

Valid Hexadecimal integer constants are:

OX0	OX1	OXF77	Oxabcd.

Invalid Hexadecimal integer constants are:

OBEF	x is not included
Ox.4bff	illegal char (.)
OXGBC	illegal char G

Maximum values these constants can have are as follows:

Integer constants	**Maximum value**
Decimal integer	32767
Octal integer	77777
Hexadecimal integer	7FFF

Unsigned integer constants: Exceed the ordinary integer by magnitude of 2, they are not negative. A character U or u is prefixed to number to make it unsigned.

Long Integer constants: These are used to exceed the magnitude of ordinary integers and are appended By L.

For example,

50000U	decimal unsigned.
1234567889L	decimal long.
0123456L	octal long.
0777777U	octal unsigned.

2.9.2 Floating Point Constants

What is a base 10 number containing decimal point or an exponent.

Examples of valid floating point numbers are:

0.	1.
000.2	5.61123456
50000.1	0.000741
1.6667E+3	0.006e-3

Examples of Invalid Floating Point numbers are:

1	decimal or exponent required.
1,00.0	comma not allowed.
2E+10.2	exponent is written after integer quantity.
3E 10	no blank space.

A Floating Point number taking the value of 5 x 10^4 can be represented as:

5000.	5e4
5e+4	5E4
5.0e+4	.5e5

The magnitude of floating point numbers range from 3.4E –38 to a maximum of 3.4E+38, through 0.0. They are taken as double precision numbers. Floating Point constants occupy 2 words = 8 bytes.

2.9.3 Character Constants

This constant is a single character enclosed in apostrophes ' ' .

For example, some of the character constants are shown below:

'A', 'x', '3', '$'

'\0' is a null character having value zero.

Character constants have integer values associated depending on the character set adopted for the computer. ASCII character set is in use which uses 7-bit code with 2^7 = 128 different characters. The digits 0-9 are having ASCII value of 48-56 and 'A' have ASCII value from 65 and 'a' having value 97 are sequentially ordered. For example,

'A' has 65, blank has 32

ESCAPE SEQUENCE

There are some non-printable characters that can be printed by preceding them with '\' backslash character. Within character constants and string literals, you can write a variety of **escape sequences**. Each escape sequence determines the code value for a single character. You can use escape sequences to represent character codes:

- you cannot otherwise write (such as \n)
- that can be difficult to read properly (such as \t)
- that might change value in different target character sets (such as \a)

- that must not change in value among different target environments (such as \0)

The following is the list of the escape sequences:

Character	Escape Sequence
"	\"
'	\'
?	\?
\	\\
BEL	\a
BS	\b
FF	\f
NL	\n
CR	\r
HT	\t
VT	\v

2.9.4 String Constants

It consists of sequence of characters enclosed within double quotes. For example,

“Red”　　“Blue Sea”　　“41213*(I+3)”.

2.10 SYMBOLIC CONSTANTS

Symbolic Constant is a name that substitutes for a sequence of characters or a numeric constant, a character constant or a string constant. When program is compiled each occurrence of a symbolic constant is replaced by its corresponding character sequence. The syntax is as follows:

```
#define name text
```

Where **name** implies symbolic name in caps.

Text implies value or the text.

For example,

```
#define printf print

#define MAX 100
```

```
#define TRUE 1

#define FALSE 0

#define SIZE 10
```

The **#** character is used for preprocessor commands. A **preprocessor** is a system program, which comes into action prior to Compiler, and it replaces the replacement text by the actual text. This will allow correct use of the statement printf.

Advantages of using Symbolic Constants are:

• They can be used to assign names to values

• Replacement of value has to be done at one place and wherever the name appears in the text it gets the value by execution of the preprocessor. This saves time. If the Symbolic Constant appears 20 times in the program; it needs to be changed at one place only.

Lesson-3 Operators

3.1 INTRODUCTION

C supports a rich set of operators. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical of logical expressions. C operators are 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

3.2 ARITHMETIC OPERATORS

C provides all the basic arithmetic operators. The operators +, -, *, / all works the same way as they do in other languages. These operators can operate on any built-in data types in C. Arithmetic operators are listed in Table 3.1

Table 3.1 Arithmetic operators

Operators	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division

Examples of arithmetic operators are

a-b a+b a*b

a%b a/b -a*b

Here a and b are variables and are known as operands. The modulo division produces the remainder of an integer division. The modulo division operator cannot be used on floating point data. C does not have any operator for exponentiation.

3.2.1Integer 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. In above examples, if a and b are integers, then for a = 13 and b = 5we have the following results:

a – b = 8 a + b = 21 a * b = 65

a % b= 3(remainder of division) a / b = 2 (decimal part truncated)

During modulo division the sign of the result is always the sign of the first operand.

That is:

-14 % 3 = -2

-14 % -3 = -2

14 % -3 = 2

3.2.2 Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. If **x and y** are floats then we will have:

x = 6.0 / 7.0 = 0.857143

y = 1.0 / 3.0 = 0.333333

The operator % cannot be used with real operands.

3.2.3 Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression and its result is always a real number. Thus

15 / 10.0 = 1.5

Whereas

15 / 10 = 1

3.3 RELATIONAL OPERATORS

In real world we always compare two or more quantities and depending upon their relation, take certain decisions. For example, we may compare the age of two person or price of two items. These comparisons can be done with the help of relational operators. The expression containing a relational operator is termed as a relational expression. The value of a relational expression is either one or zero. It is one if the specified relation is true and zero if the relation is false. For Example:

10 < 25 is true

25 < 10 is false

C Supports six relational operators in all. The operators and their meaning are shown in Table 3.2

Table 3.2 Relational operators

Operators	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

A simple relational expression contains only one relational operator and takes the following form:

ae1 relational operator ae2

ae1 and ae2 are arithmetic expressions, which may be simple constants, variable or combination of them. Examples of simple relational expression are given below:

```
4.9  <=  10         True
4.9  <    -10                False
-34  >=  0          False
10  <   7+5         True
```

When arithmetic expressions are used on either side of a relational operator, the arithmetic expression will be evaluated first and then the results compared.

3.4 LOGICAL OPERATORS

C has the following three logical operators.

Table 3.3 Logical operators

Operators	Meaning
&&	logical **AND**
||	logical **OR**
!	logical **NOT**

The logical operators && and || are used when we want tot test more than one conditions 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. Like the simple relational expression, a logical expression also yields a value of one or zero according to the Table 3.4. The logical expression given above is true only if a>b is true and x = = 10 is true. If either (or both) of them are false, the expression is false.

Table 3.4 Truth table

op1	op2	Value of expression	
		op1 && op2	op1 || op2
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Example of logical operators:

```
if(age>55 && sal<1000)

if(number<0 || number>100)
```

3.5 ASSIGNMENT OPERATORS

Assignment operators are used to assign the results of the expression to the variable. The usual assignment operator is '='. In addition, C has a set of 'shorthand' assignment operators of the form

v **op** = exp;

where v is a variable, exp is an expression and op is a C binary arithmetic operator. The operator op= is known as shorthand assignment operator.

The assignment statement

v **op** = exp;

is equivalent to

v= v **op (** exp) ;

Consider an example:

x += y+1;

This is same as the statement

x=x+(y+1);

3.5.1 Shorthand Assignment Operators

Table 3.5 Shorthand assignment operators

Shorthand operators	simple assignment operators
a + =1	a = a + 1
a - = 1	a = a – 1
a *= n + 1	a = a * (n+1)
a /= n + 1	a = a / (n+1)
a %= b	a = a % b

3.6 INCREMENT AND DECREMENT OPERATORS

C has two very useful operators that are not generally found in other languages. These are the increment and decrement operator:

++ and --

The operator ++ adds 1 to the operands while – subtracts 1.It takes the following form:

```
++m; or m++;

--m; or m--;
```

Use of increment and decrement operators are with for and while loops extensively. While ++m and m++ means the same thing when they form the statements independently, they behave differently when they are used in expression on the right hand side of an assignment statement. Consider the following:

```
m = 5;

y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we write the above statement as

```
m = 5;

y = m++;
```

Then, the value of y would be 5 and m would be 6. A prefix (++m) operator first adds 1 to the operand and then the result is assigned to the variable on left. On other hand a postfix (m++) operator first assign the value to the variable on the left and then increment the operand.

3.7 CONDITIONAL OPERATOR

A ternary operator pair "? : " is available in C to construct conditional expression of the form:

```
exp1 ? exp2 : exp3;
```

Here exp1 is evaluated first. If it is true then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false then exp3 is evaluated and its value becomes the value of the expression. Only one of the expressions (either exp2 or exp3) is evaluated. For example, consider the following statements:

```
        a = 12;

        b = 15;

        x = ( a >b ) ? a : b ;
```

In this example, x will be assigned the value of b. this can be achieved using if… else statements as follow:

```
    if(a > b)

        x = a;

    else

        x = b;
```

3.8 BITWISE OPERATORS

C has distinction on supporting special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. Table 3.6 lists the bitwise operators and their meaning.

Table 3.6 Bitwise operators

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Shift left
>>	Shift right
~	One's complement

3.9 SPECIAL OPERATORSC supports some special operators such as

- Comma operator
- Size of operator

3.9.1 The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression. For example:

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

This statement first assigns the value 10 to **x**, then assigns 5 to **y,** and finally assigns 15 (i.e,10+5) to **value**.

3.9.2 The Size of Operator

The size of is a compiler time operator and, when used with an operand, it returns the number of bytes the operand occupies.

Examples:

m = **sizeof**(sum);

n = **sizeof(long int)**

k = **sizeof**(235L)

Lesson-4 Expression

4.1 INTRODUCTION

Expressions in C are basically operators acting on operands. Statements like a = b + 3, ++z, a-- and 300 > (8 * k) are all expressions. Strictly speaking, even a single variable or constant can be considered an expression. You have seen several expressions in the previous C tutorial on Operators in which the examples involved expressions.

4.2 Arithmetic Instruction

A C arithmetic instruction consists of a variable name on the left hand side of = and variable names & constants on the right hand side of =. The variables and constants appearing on the right hand side of = are connected by arithmetic operators like **+, -, *,** and **/**.

Example:-

```
 int ad ;

float kot, deta, alpha, beta, gamma ;

ad = 3200 ;

kot = 0.0056 ;

deta = alpha * beta / gamma + 3.2 * 2 / 5 ;
```

Here,

***, /, -, +** are the arithmetic operators.

= is the assignment operator.

2, 5 and 3200 are integer constants.

3.2 and 0.0056 are real constants.

ad is an integer variable.

kot, deta, alpha, beta, gamma are real variables.

The variables and constants together are called 'operands' that are operated upon by the 'arithmetic operators' and the result is assigned, using the assignment operator, to the variable on left-hand side.

A C arithmetic statement could be of three types. These are as follows:

4.2.1 Integer mode arithmetic statement:

This is an arithmetic statement in which all operands are either integer variables or integer constants.

Ex.: int i, king, issac, noteit ;

i = i + 1 ;

king = issac * 234 + noteit - 7689 ;

4.2.2 Real mode arithmetic statement

This is an arithmetic statement in which all operands are either real constants or real variables.

Ex.: float qbee, antink, si, prin, anoy, roi ;

qbee = antink + 23.123 / 4.5 * 0.3442 ;

si = prin * anoy * roi / 100.0 ;

4.2.3 Mixed mode arithmetic statement

This is an arithmetic statement in which some of the operands are integers and some of the operands are real.

Ex.: float si, prin, anoy, roi, avg ;

int a, b, c, num ;

si = prin * anoy * roi / 100.0 ;

avg = (a + b + c + num) / 4 ;

It is very important to understand how the execution of an arithmetic statement takes place. Firstly, the right hand side is evaluated using constants and the numerical values stored in the variable names. This value is then assigned to the variable on the left-hand side.

4.3 Integer and Float Conversions

In order to effectively develop C programs, it will be necessary to understand the rules that are used for the implicit conversion of floating point and integer values in C. These are mentioned below. Note them carefully.

(a) An arithmetic operation between an integer and integer always yields an integer result.

(b) An operation between a real and real always yields a real result.

(c) An operation between an integer and real always yields a real result. In this operation the integer is first promoted to a real and then the operation is performed. Hence the result is real.

I think a few practical examples shown in the Table 4.1 would put the issue beyond doubt.

Table 4.1 Integer and Float Conversions

Operation	Result	Operation	Result
5 / 2	2	2 / 5	0
5.0 / 2	2.500000	2.0 / 5	0.400000
5 / 2.0	2.500000	2 / 5.0	0.400000
5.0 / 2.0	2.500000	2.0 / 5.0	0.400000

4.4 Type Conversion in Assignments

It may so happen that the type of the expression and the type of the variable on the left-hand side of the assignment operator may not be same. In such a case the value of the expression is promoted or demoted depending on the type of the variable on left-hand side of =.

For example, consider the following assignment statements.

```
int i ;

float b ;

i = 3.5 ;

b = 30 ;
```

Here in the first assignment statement though the expression's value is a **float** (3.5) it cannot be stored in **i** since it is an **int**. In such a case the **float** is demoted to an **int** and then its value is stored. Hence what gets stored in **i** is 3. Exactly opposite happens in the next statement. Here, 30 is promoted to 30.000000 and then stored in **b**, since **b** being a **float** variable cannot hold anything except a **float** value.

Instead of a simple expression used in the above examples if a complex expression occurs, still the same rules apply. For example, consider the following program fragment.

```
float a, b, c ;

int s ;

s = a * b * c / 100 + 32 / 4 - 3 * 1.1 ;
```

Here, in the assignment statement some operands are **int**s whereas others are **float**s. As we know, during evaluation of the expression the **int**s would be promoted to **float**s and the result of the expression would be a **float**. But when this **float** value is assigned to **s** it is again demoted to an **int** and then stored in **s**.

Observe the results of the arithmetic statements shown in Table 4.2. It has been assumed that **k** is an integer variable and **a** is a real variable.

Table 4.2 Type Conversion in Assignments

Arithmetic Instruction	Result	Arithmetic Instruction	Result
k = 2 / 9	0	a = 2 / 9	0.000000
k = 2.0 / 9	0	a = 2.0 / 9	0.222222
k = 2 / 9.0	0	a = 2 / 9.0	0.222222
k = 2.0 / 9.0	0	a = 2.0 / 9.0	0.222222
k = 9 / 2	4	a = 9 / 2	4.000000
k = 9.0 / 2	4	a = 9.0 / 2	4.500000
k = 9 / 2.0	4	a = 9 / 2.0	4.500000
k = 9.0 / 2.0	4	a = 9.0 / 2.0	4.500000

Note that though the following statements give the same result, 0, the results are obtained differently.

k = 2 / 9 ;

k = 2.0 / 9 ;

In the first statement, since both 2 and 9 are integers, the result is an integer, i.e. 0. This 0 is then assigned to **k**. In the second statement 9 is promoted to 9.0 and then the division is performed. Division yields 0.222222. However, this cannot be stored in **k**, **k** being an **int**. Hence it gets demoted to 0 and then stored in **k**.

4.5 Hierarchy of Operations

While executing an arithmetic statement, which has two or more operators, we may have some problems as to how exactly does it get executed. For example, does the expression 2 * x - 3 * y correspond to (2x)-(3y) or to 2(x-3y)? Similarly, does A / B * C correspond to A / (B * C) or to (A / B) * C? To answer these questions satisfactorily one has to understand the 'hierarchy' of operations. The priority or precedence in which the operations in an arithmetic statement are performed is called the hierarchy of operations. The hierarchy of commonly used operators is shown in Table 4.3.

Table 4.3 Hierarchy of commonly used operators

Priority	Operators	Description
1st	* / %	Multiplication, division, modular division
2nd	+ -	addition, subtraction
3rd	=	assignment

Following example would clarify the issue further.

Example Determine the hierarchy of operations and evaluate the following expression:

i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8

Stepwise evaluation of this expression is shown below:

```
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8
i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8          operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8              operation: /
i = 1 + 1+ 8 - 2 + 5 / 8                   operations:/
i = 1 + 1 + 8 - 2 + 0                      operation: /
i = 2 + 8 - 2 + 0                          operation: +
i = 10 - 2 + 0                             operation: +
i = 8 + 0                                  operation: -
i = 8                                      operations: +
```

Note that 6 / 4 gives 1 and not 1.5. This so happens because 6 and 4 both are integers and therefore would evaluate to only an integer constant. Similarly 5 / 8 evaluates to zero, since 5 and 8 are integer constants and hence must return an integer value.

All operators in C are ranked according to their precedence. And mind you there are as many as 45 odd operators in C, and these can affect the evaluation of an expression in subtle and unexpected ways if we aren't careful. Unfortunately, there are no simple rules that one can follow, such as “BODMAS” that tells algebra students in which order does an expression evaluate. We have not

encountered many out of these 45 operators, so we won’t pursue the subject of precedence any further here. However, it can be realized at this stage that it would be almost impossible to remember the precedence of all these operators. So a full-fledged list of all operators and their precedence is given in Appendix A. This may sound daunting, but when its contents are absorbed in small bites, it becomes more palatable.

So far we have seen how the computer evaluates an arithmetic statement written in C. But our knowledge would be incomplete unless we know how to convert a general arithmetic statement to a C statement. C can handle any complex expression with ease. Some of the examples of C expressions are shown in Table 4.4.

Table 4.4 Algebraic Expression and its equivalent C Expression

Algebric Expression	C Expression
a x b – c x d	a * b – c * d
(m + n) (a + b)	(m + n) * (a + b)
$3x^2 + 2x + 5$	3 * x * x + 2 * x + 5
	(a + b + c) / (d + e)
	2 * b * y / (d + 1) – x / 3 * (z + y)

4.6 Associativity of Operators

When an expression contains two operators of equal priority the tie between them is settled using the associativity of the operators. Associativity can be of two types—Left to Right or Right to Left. Left to Right associativity means that the left operand must be unambiguous. Unambiguous in what sense? It must not be involved in evaluation of any other sub-expression. Similarly, in case of Right to Left associativity the right operand must be unambiguous. Let us understand this with an example.

Consider the expression

a = 3 / 2 * 5 ;

Here there is a tie between operators of same priority, that is between / and *. This tie is settled using the associativity of / and *. But both enjoy Left to Right associativity. Figure 4.1 shows for each operator which operand is unambiguous and which is not.

Figure 4.1 Associatively of Operators

Operator	Left	Right	Remark
/	3	2 or 2 * 5	Left operand is unambiguous, Right is not
*	3 / 2 or 2	5	Right operand is unambiguous, Left is not

Since both / and * have L to R associativity and only / has unambiguous left operand (necessary condition for L to R associativity) it is performed earlier.

Consider one more expression

a = b = 3 ;

Here both assignment operators have the same priority and same associativity (Right to Left). Figure 4.2 shows for each operator which operand is unambiguous and which is not.

Figure 4.2 Associatively of Operators

Operator	Left	Right	Remark
=	a	b or b = 3	Left operand is unambiguous, Right is not
=	b or a = b	3	Right operand is unambiguous, Left is not

Since both = have R to L associativity and only the second = has unambiguous right operand (necessary condition for R to L associativity) the second = is performed earlier.

Consider yet another expression

z = a * b + c / d ;

Here * and / enjoys same priority and same associativity (Left to Right). Figure 4.3 shows for each operator which operand is unambiguous and which is not.

Figure 4.3 Associatively of Operators

Operator	Left	Right	Remark
*	A	b	Both operands are unambiguous
/	C	d	Both operands are unambiguous

Here since left operands for both operators are unambiguous Compiler is free to perform * or / operation as per its convenience since no matter which is performed earlier the result would be same.

Lesson-5 Managing Input and Output Operations

5.1 INTRODUCTION

In C language reading, writing and processing of data are three important functions. Most programs take some data as input and display the processed data, often known as information or result, on a suitable medium. So far we have seen two methods of providing data to the program variables. One method is to assign value to the variable through the assignment statement such as **a=10 , x = 6**; and so on. Another method is to use input function **scanf** which can read data from keyboard. For outputting the results we have used the function **printf** which sends the results to display device. Unlike other languages C does not have any built-in input/output statement as a part of syntax. All input and output operations are carried out through functions such as **printf** and **scanf**. There exist some functions that have become standards for input and out operations in C. These functions are collectively known as standard I/O library. In C language, any program that uses a standards input/output function must contain the statement **#include<stdio.h>** at the beginning. The file name **stdio.h** is abbreviation for *standard input-output header* file. The instruction **#include<stdio.h>** tells the compiler ' to search the file named **stdio.h** and place its content at this point in program'. The content of the header file become the part of the code when it is compiled.

5.2 Reading a character

The simplest of all input/output operations is reading a character from the standard input device (keyboard) and writing it to the standard output device (screen). Reading a single character can be done using the function **getchar().** This can also be done with the help of the scanf function.

The **getchar** takes the following form:

variable_name = getchar();

variable_name is valid C name that has been declared as **char** type. When this statement is encountered, the computer will waits untill a key is pressed and then assigns the character as a value to **getchar** function. Since **getchar** is used on right hand side of assignment statement, the character value of **getchar** is in turn assigned to the variable on the left. For example:

char ch;

ch = getchar();

will assign "g" to the variable **ch** when we press the key g on the keyboard.

The getchar function may be called successively to read the characters in a line of text. For example, the following program segment reads characters from keyboard one after the another until the return key is pressed

```
- - - - - - - - - -

- - - - - - - - - -

- - - - - - - - - -

char ch;

ch=' ';

while(ch!= '\n')

{

        ch = getchar();

}

- - - - - - - - - -

- - - - - - - - - -

- - - - - - - - - -
```

5.3 Writing a character

Like **getchar,** there is a function called **putchar** for writing a character one at a time to the screen, it takes the form as shown below:

putchar(variable_name)

where variable_name is a type **char** variable containing character. This statement display the character contained in variable_name at the screen. For example the statements:

```
ans =  'y';

putchar(ans);
```

will display the character Y on the screen. The statement **putchar('\n')** would cause the curser on the screen to move to the beginning of the next line.

5.4 FORMATTED INPUT

Formatted input refers to an input data that has been arranged in a particular format. The formatted data can be read with the help of **scanf** function. The general form of **scanf** is

scanf("control string",arg1,arg2….argn);

The control string specifies the field format in which the data is to be entered and the arguments arg1,arg2...argn specifies the address of locations where the data are stored. Control strings and arguments are separated by commas. Control string contains field specification, which directs the interpretation of input data. It may include:

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

5.4.1 Inputting integer numbers

The field specification for reading an integer number is

%w d

The percentage sign (%) indicates 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, the number to be read is in integer mode. Consider the following example:

scanf("%2d %5d",&num1, &num2);

Data line:

```
52    31456
```

The value 50 is assigned to **num1** and 31456 to **num2**. Suppose the input data is as follows:

```
31456   52
```

The **num1** will assign 31 (because %2d) and **num2** will be assigned 456 (unread part of 31456). The value 50 that is unread will be assigned to the first variable in the next **scanf** call. This kind of error may be eliminated if we use the field specification without the field width. That is the statement:

scanf("%d %d",&num1, &num2);

will read the data

```
31456   52
```

correctly and assign 31456 to **num1** and 52 to **num2**.

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

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

will assign data

```
121  456  789
```

As follow:

121 to **a**

456 **skipped (because of *)**

789 to **b**

Program

```
/*Reading integer numbers*/

#include<stdio.h>

#include<conio.h>

void main()

{

int a, ,b, c, x, y, z;

int p, q, r;

printf("Enter three integer numbers \n");

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

printf("%d %d %d \n \n",a, b, c);

printf("Enter two 4-digit numbers \n");

scanf("%2d %4d ",&x, &y);

printf("%d %d \n \n",x, y);

printf("Enter two integer numbers \n");

scanf("%d %d",&a, &x);

printf("%d %d \n \n",a, x);

printf("Enter a nine digit numbers \n");

scanf("%3d %4d %3d",&p, &q, &r);

printf("%d %d %d \n \n",p, q, r);

printf("Enter two three digit numbers \n");

scanf("%d %d",&x, &y);

printf("%d %d \n \n",x, y);

}
```

OUTPUT

```
Enter three integer numbers
```

```
1 2 3

1 3 –3577

Enter two 4-digit numbers

6789 4321

67 89

Enter two integer numbers

44 66

4321 44

Enter a nine digit numbers

123456789

66 1234 567

Enter two three digit numbers

123 456

89 123
```

5.4.2 Inputting Real Numbers

Unlike integer numbers, the field width of real numbers is not to be specified and therefore **scanf** reads real numbers using simple specification **%f** for both the notations, namely, decimal point notation and exponential notation. For example the statement:

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

with the input data

```
435.25  42.31E-1  687
```

Will assign the value 435.25 to **x,** 4.231to **y,** and 687.0 to **z.**

If the number to be read is of **double** type, then the specification should be **%lf** instead of simple **%f**.

5.4.3Inputting Character Strings

Following are the specifications for reading character strings:

%ws or **%wc**

Some versions of **scanf** support the following conversion specifications for strings:

%[characters] and %[^characters]

The specification **%[characters]** means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification **%[^characters]** does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string.

5.4.4 Reading Mixed Data Types

It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, it should be ensured that the input data items match the control specifications in order and type.for example:

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

Table 5.1 will show some of the commonly used **scanf** formatted codes:

Table 5.1 commanly used Scanf Format Codes

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
%[..]	Read a string of word(s)

Points To Remember while using scanf

- All function arguments, except the control string, must be pointers to variables (use &).
- Format specifications contained in the control string should match the arguments in order.
- Input data items must be separated by spaces and must match the variables receiving the input in the same order.
- The reading will be terminated, when scanf encounters an 'invalid mismatch' of data or a character that is not valid for the value being read.
- When searching for a value, scanf ignores line boundaries and simply looks for the next appropriate character.
- Any unread data items in a line will be considered as a part of the data input line to the next scanf call.
- When the field width specifier w is used, it should be large enough to contain the input data size.

5.5 FORMATTED OUTPUT

The **printf** statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the screen. The general form of **printf** statement is

printf("control string",arg1, arg2…argn);

Control string consists of three types of items:

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

The control string indicates how many argument follows and what their types are. The arguments arg1, arg2, …, argn are the variables whose values are formatted and printed according to the specification of the control string. The arguments should match in number, order and type with the format specifications.

A simple formate specification has the following form:

% w.p type-spcifier

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 number of digits to right of the decimal point for real number. Both **w** and **p** are optional.

5.5.1 Output of Integer Numbers

The format specification for printing an integer number is

%w d

where **w** specifies the minimum field width for the output. However, if the number is greater than the specified field width, it will be printed in full, overriding the minimum specification. **d** specifies the value to be printed is an integer. The number is right-justified in the given field width. Leading blanks will appear as necessary. The following example will output the number 6789 under different formats:

Format	Output
printf("%d",6789);	`6789`
printf("%6d",6789);	`  6789`
printf("%2d",6789);	`6789`
printf("%-6d",6789);	`6789  `
printf("%06d",6789);	`006789`

5.5.2 Output of Real Numbers

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

%w.p f

The integer **w** indicates the minimum number of positions that are to be used for the display of the value and the integer **p** indicates the number of digits to be displayed after the decimal point.

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

%w.p e

The following example will illustrate the output of the number y = 56.4567 under different format specification:

Format	Output
printf("%7.4f",y);	`56.4567`
printf("%7.2f",y);	`  56.46`
printf("%-7.2f",y);	`56.46  `
printf("%f",y);	`56.4567`
printf("%10.2e",y);	`  5.64E+01`
printf("%11.4 e",-y);	`-5.6456E+01`
printf("%-10.2e",y);	`5.64E+01  `
printf("%e",y);	`5.645670E+01`

5.5.3 Printing of 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. The default value for w is 1.

5.5.4 Printing of Strings

The format specification for outputting strings is of the form:

%w.ps

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

5.5.5 Mixed Data Output

It is permitted to mix data types in one printf statements. For example the statement of the following type

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

is valaid. As printf uses its control string to decide how many variables to be printed and what their types are. Therefore, the formate specification should match the variable in number, order and type.

Table 5.1 will show some of the commonly used **scanf** formatted codes:

Table 5.1 commanly used Scanf Format Codes

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
%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

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

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

Table 5.3 will show some of the commonly used **output** formatted flags:

Table 5.3 commanly used output Format flags

Flag	Meaning
-	Output is left justified with in the field. Remaining field will be blank.
+	+ or – will precede the sign numeric item.
#(with o or x)	Causes octal and hex items to be presented by O and Ox, respectib\vely.
#(with e, f or g)	Causes the decimal point to be presented in all floating point numbers, even if it is whole number. Also prevents the truncation of trailing zero in g-type conversion.

Module-2 Decision Making, Looping and Functions

Lession- 6 Decision Making

6.1 CONTROL STATEMENTS

The C language programs presented until now follows a sequential form of execution. Many times it is required to alter the flow of the sequence of instructions. C language provides statements that can alter the flow of a sequence of instructions. These statements are called control statements. These statements help to jump from one part of the program to another. The control transfer may be conditional or unconditional.

C language supports the following statements known as *control* or *decision making* statements.

1. **if** statement

2. **switch** statement

3. Conditional operator statement

4. **goto** statement

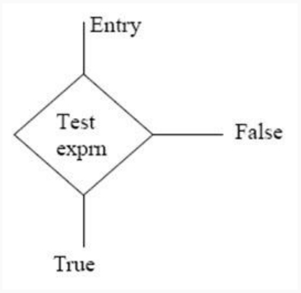
6.2 CONDITIONAL STATEMENTS

IF STATEMENT

The **if** statement is used to control the flow of execution of statements and isof the form:

If(test expression)

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression is 'true' or 'false', it transfers the control to a particular statement.



Eg: **if**(bank balance is zero)

Borrow money

The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested.

1. Simple **if** statement

2. **if**…..**else** statement

3. Nested **if**…..**else** statement

4. **elseif ladder**

6.2.1 SIMPLE IF STATEMENT

The general form of a simple **if** statement is The 'statement-block' may be a single statement or a group of statement. If the test expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x.

If(test exprn)

{

statement-block;

}

statement-x;

Example:

if(category = = SPORTS)

{

marks = marks + bonus_marks;

}

printf("%f ",marks);

……….

……….

6.2.2 THE IF…ELSE STATEMENT

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

If(test expression)

{

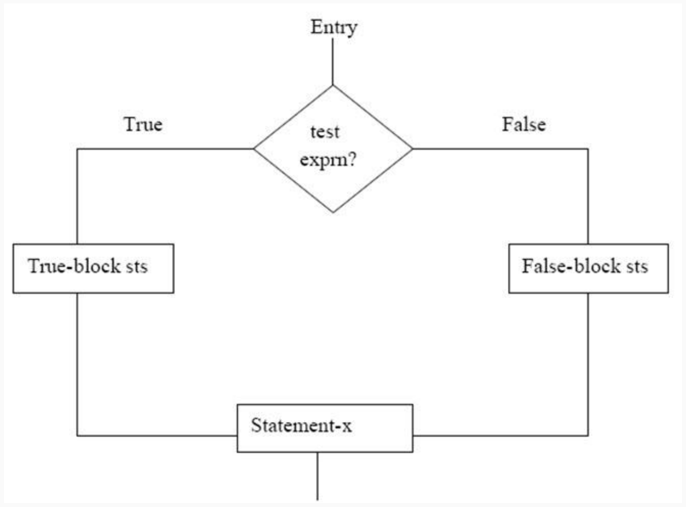
True-block statement(s);

}

else

```
{
False-block statement(s);
}
statement-x;
```

If the test expression is true, then the true block statements are executed; otherwise the false block statement will be executed.



For Example:

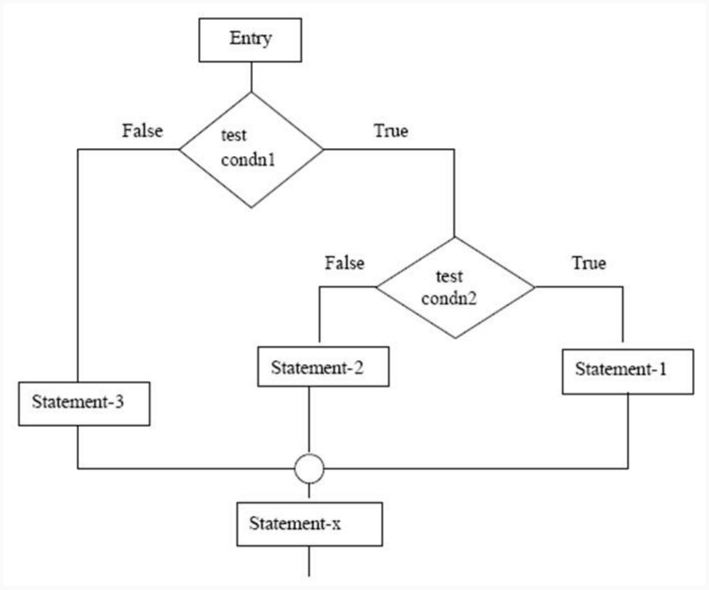
```
………
………
if(code ==1)
boy = boy + 1;
if(code == 2)
girl =girl + 1;
………
………
```

6.2.3 NESTING OF IF…..ELSE STATEMENTS

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

```
If(test condition 1)
{
```

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



Program

```
/*Selecting the largest of three values*/
void main()
{
float A, B, C;
printf("Enter three values \n");
scanf(" %f %f %f ",&A, &B, &C);
printf("\nLargest value is:");
if(A > B)
{
if(A > C)
printf(" %f \n",A);
else
printf(" %f \n",C);
}
else
{
if(C > B)
printf(" %f \n",C);
else
printf(" %f \n",B);
}
}
```

OUTPUT

```
Enter three values:
5 8 24
Largest value is 24
```

6.2.4 THE ELSEIF LADDER

The general form is

If(condn 1)

Statement-1;

else if (condn 2)

statement-2;

else if (condn 3)

statement-3;

……….

……….

else if (condn n)

statement-n;

else

default statement;

statement-x;

Program

```
/*Use of else if ladder*/
void main()
{
int units, cno;
float charges;
printf("Enter customer no. and units consumed \n");
scanf("%d %d",&cno, &units );
if(units <= 200)
charges = 0.5 * units;
else if(units <= 400)
charges = 100+ 0.65 * (units – 200)
else if (units <= 600)

charges = 230 + 0.8 * (units – 400)

else

charges = 390 + (units – 600);

printf("\n \ncustomer no =%d,charges =%.2f \n",cno,charges);

}
```

OUTPUT

```
Enter customer no. and units consumed 101 150

Customer no=101 charges = 75.00
```

6.3 THE SWITCH STATEMENT

Switch statement is used for complex programs when the number of alternatives increases. The switch statement tests the value of the given variable against the list of **case** values and when a match is found, a block of statements associated with that case is executed.The general form of switch statement is:

```
switch ( integer expression )

{

case constant 1 :

do this ;

case constant 2 :

do this ;

case constant 3 :

do this ;

default :

do this ;

}
```

The integer expression following the keyword **switch** is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an integer. The keyword **case** is followed by an integer or a character constant. Each constant in each **case** must be different from all the others. The "do this" lines in the above form of **switch** represent any valid C statement.

What happens when we run a program containing a **switch?** First, the integer expression following the keyword switch is evaluated. The value it gives is then matched, one by one, against the constant values that follow the **case** statements. When a match is found, the program executes the statements following that **case**, and all subsequent **case** and **default** statements as well. If no match is found with any of the **case** statements, only the statements following the **default** are executed. Following examples will show how this control structure works.

```
………
………
index = marks / 10;
switch(index)
{
case 10:
case 9:
case 8:
grade = “Honours”;
break;
case 7:
case 6:
grade = “first division”;
break;
case 5:
grade = “second division”;
break;
case 4:
grade = “third division”;
break;
default:
grade = “first division”;
break;

}

printf(“%s \n”,grade);
```

If you want that only case 5 should get executed, it is upto you to get out of the **switch** then and there by using a **break** statement.

6.4 THE ?: OPERATOR

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of **?** and **:** and takes three operands. It is of the form:

exp1?exp2:exp 3

Here exp1 is evaluated first. If it is true then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false then exp3 is evaluated and its value becomes the value of the expression. For example:

```
if(x < 0)

flag = 0;

else

flag = 1;
```

can be written as:

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

6.5 THE GOTO STATEMENT

C supports the goto statement to branch unconditionally from one point of the program to nother. 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 general from is:

```
goto label                     label:

----------                     statement;

----------                     ----------

label:                         ----------

statement;                      goto label
```

Note: A label can be anywhere in the program, either before or after the **goto** label;statement.

Example of goto label statement:

```
void main()

{

double x, y;

read:

scanf("%f",&x);

if(x < 0)

goto read;

y = sqrt(x);

printf("%f %f \n",x, y);

goto read;
```

Lesson-7 Branching and Looping

7.1 INTRODUCTION

The programs that we have developed so far used either a sequential or a decision control instruction. In the first one, the calculations were carried out in a fixed order, while in the second, an appropriate set of instructions were executed depending upon the outcome of the condition being tested (or a logical decision being taken).

These programs were of limited nature, because when executed, they always performed the same series of actions, in the same way, exactly once. Almost always, if something is worth doing, it's worth doing more than once. You can probably think of several examples of this from real life, such as eating a good dinner or going for a movie. Programming is the same; we frequently need to perform an action over and over, often with variations in the details each time. The mechanism, which meets this need, is the 'loop', and loops are the subject of this chapter.

7.2 Loops

The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive operation is done through a loop control instruction.

There are three methods by way of which we can repeat a part of a program. They are:

1. Using a **while** statement
2. Using a **do-while** statement
3. Using a **for** statement

A program loop therefore consists of two segments:

- Body of the loop
- control statement

The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop. In looping process in general 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 conditions for the execution of the loop

4. Incrementing the counter

7.2.1 While Statement

When in a program a single statement or a certain group of statements are to be executed repeatedly depending upon certain test condition, then while statement is used.

The syntax is as follows:

```
while (test condition)

{

body_of_the_loop;

}
```

Here, test condition is an expression that controls how long the loop keeps running. Body of the loop is a statement or group of statements enclosed in braces and are repeatedly executed till the value of test condition evaluates to true. As soon as the condition evaluates to **false**, the control jumps to the first statement following the while statement. If condition initially itself is **false**, the body of the loop will never be executed. While loop is sometimes called as entry-control loop, as it controls the execution of the body of the loop depending upon the value of the test condition. This is shown in the figure 7.1 given below:

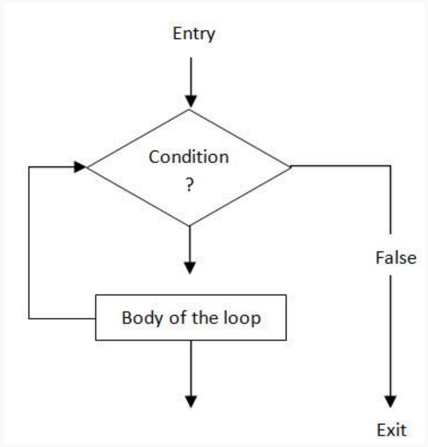


Figure 7.1: The while loop statement

Let us consider a program to illustrate while loop,

Write a program to calculate the factorial of a given input natural number.

```
/* Program to calculate factorial of given number */

#include <stdio.h>

#include <math.h>

#include <stdio.h>
```

```
void main( )

{

int x;

long int fact = 1;

printf("Enter any number to find factorial:\n"); /*read the number*/

scanf("%d",&x);

while (x > 0)

{

fact = fact * x; /* factorial calculation*/

x=x-1;

}

        printf("Factorial is %ld",fact);

}
```

7.2.2 The do...while Loop

There is another loop control structure which is very similar to the while statement – called as the do.. while statement. The only difference is that the expression which determines whether to carry on looping is evaluated at the end of each loop. The syntax is as follows:

```
do

{

statement(s);

} while(test condition);
```

In do-while loop, the body of loop is executed at least once before the condition is evaluated. Then the loop repeats body as long as condition is true. However, in while loop, the statement doesn't execute the body of the loop even once the condition is false. That is why do-while loop is also called exit-control loop. This is shown in the figure 7.2 given below.

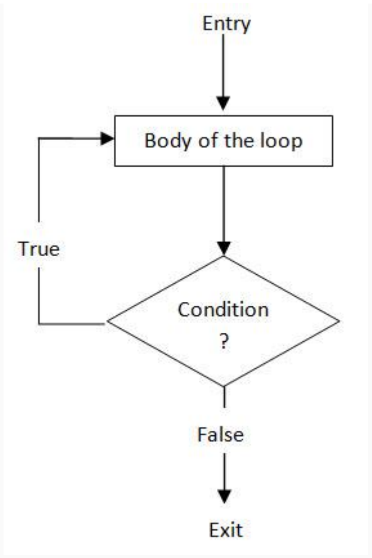


Figure 7.2: The do…while statement

Let us consider a program to illustrate do..while loop,

Write a program to print first ten even natural numbers.

```
/* Program to print first ten even natural numbers */

#include <stdio.h>

void main()

{

int i=0;

int j=2;

do

{

printf("%d",j);

j =j+2;

i=i+1;

} while (i<10);

}
```

OUTPUT

```
2 4 6 8 10 12 14 16 18 20
```

7.2.3 The for Loop

for statement makes it more convenient to count iterations of a loop and works well where the number of iterations of the loop is known before the loop is entered. The syntax is as follows:

```
for (initialization; test condition; increment or decrement)

{

Statement(s);

}
```

The main purpose is to repeat statement while condition remains true, like the while loop. But in addition, for provides places to specify an initialization instruction and an increment or decrement of the control variable instruction. So this loop is specially designed to perform a repetitive action with a counter.

The for loop as shown in figure 7.3, works in the following manner:

- Initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
- Condition is checked, if it is true the loop continues, otherwise the loop finishes and statement is skipped.
- Statement(s) is/are executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets { }.
- Finally, whatever is specified in the increment or decrement of the control variable field is executed and the loop gets back to step 2.

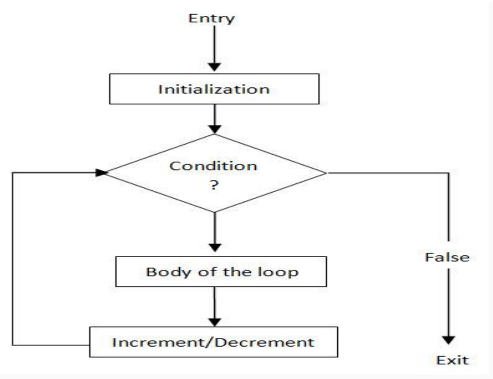


Figure 7.3: The for statement

Let us consider a program to illustrate for loop, Write a program to print first n natural numbers.

```
/* Program to print first n natural numbers */

#include <stdio.h>

void main( )

{

int i,n;

printf("Enter value of n \n");

scanf("%d",&n);

printf("\nThe first %d natural numbers are :\n", n);

for (i=1;i<=n;++i)

{

printf("%d",i);

}

}
```

OUTPUT

```
Enter value of n

6

The first 6 natural numbers are:

1 2 3 4 5 6
```

The three statements inside the braces of a for loop usually meant for one activity each, however any of them can be left blank also. More than one control variables can be initialized but should be separated by comma.

Various forms of loop statements can be:

(a). for(;condition;increment/decrement)

```
            body;
```

A blank first statement will mean no initialization.

(b). for (initialization;condition;)

body;

A blank last statement will mean no running increment/decrement.

(c). for (initialization;;increment/decrement)

body;

A blank second conditional statement means no test condition to control the exit from the loop. So, in the absence of second statement, it is required to test the condition inside the loop otherwise it results in an infinite loop where the control never exits from the loop.

(d). for (;;increment/decrement)

body;

Initialization is required to be done before the loop and test condition is checked inside the loop.

(e). for (initialization;;)

body;

Test condition and control variable increment/decrement is to be done inside the body of the loop.

(f). for (;condition;)

body;

Initialization is required to be done before the loop and control variable increment/decrement is to be done inside the body of the loop.

(g). for (;;;)

body;

Initialization is required to be done before the loop, test condition and control variable increment/decrement is to be done inside the body of the loop.

7.2.4 The Nested Loops

C allows loops to be nested, that is, one loop may be inside another. The program given below illustrates the nesting of loops.

Let us consider a program to illustrate nested loops,

Write a program to generate the following pattern given below:

1

1 2

```
1 2 3

1 2 3 4

1 2 3 4 5

/* Program to print the pattern */

#include <stdio.h>

void main( )

{

int i,j;

for (i=1;i<=4;++i)

{

printf("%d\n",i);

for(j=1;j<=i;++j)

{

printf("%d\t",j);

                        }

}

}
```

Here, an inner for loop is written inside the outer for loop. For every value of i, j takes the value from 1 to i and then value of i is incremented and next iteration of outer loop starts ranging j value from 1 to i.

Lesson -8 User Defined Function

8.1 INTRODUCTION

One of the main strength of C language is the use of functions. They are easy to define and use. We have used functions in every program that we have discussed so far. However, they have been primarily limited to three functions, namely **main**, **printf** and **scanf**. C functions can be classified into two categories:

- User defined functions
- Library functions

main is an example of user defined functions. **printf** and **scanf** belongs to the category library functions. The main difference between these two types functions is that library functions are not required to be written by us whereas user defined functions has to be developed and written by the user.

8.2 DEFINITION OF FUNCTION

Function is a self-contained program segment that carries out some specific well-defined task. Every C program consists of one or more functions. The most important function is **main**(). Program execution will always begin by carrying out the instruction in **main**. The definitions of functions may appear in any order in a program file because they are independent of one another. A function can be executed from anywhere within a program. Once the function has been executed, control will be returned to the point from which the function was accessed. Functions contain special identifiers called parameters or arguments through which information is passed to the function and from functions information is returned via the return statement. It is not necessary that every function must return information, there are some functions also which do not return any information for example the system defined function **printf**.

Before using any function it must be defined in the program. Function definition has three principal components:

- First line
- Parameter declarations
- Body of the functions

The first line of a function definition contains the data type of the information return by the function, followed by function name, and a set of arguments or parameters, separated by commas and enclosed in parentheses. The set of arguments may be skipped over. The data type can be omitted if the function returns an integer or a character. An empty pair of parentheses must follow the function name if the function definition does not include any argument or parameters.

The general term of first line of functions can be written as:

data-type function-name (formal argument 1, formal argument 2…formal argument n)

The formal arguments allow information to be transferred from the calling portion of the program to the function. They are also known as parameters or formal parameters. These formal arguments are called actual parameters when they are used in function reference. The names of actual parameters and formal parameters may be either same or different but their data type should be same. All formal arguments must be declared after the definition of function. The remaining portion of the function definition is a compound statement that defines the action to be taken by the function. This compound statement is sometimes referred to as the body of the function. This compound statement can contain expression statements, other compound statements, control statements etc. Information is returned from the function to the calling portion of the program via the return statement. The return statement also causes control to be returned to the point from which the function was accessed. In general terms, the return statement is written as:

return expression;

The value of the expression is returned to the calling portion of the program. The return statement can be written without the expression. Without the expression, return statement simply causes control to revert back to the calling portion of the program without any information transfer. The point to be noted here is that only one expression can be included in the return statement. Thus, a function can return only one value to the calling portion of the program via return. But a function definition can include multiple return statements, each containing a different expression. Functions that include multiple branches often require multiple returns. It is not necessary to include a return statement altogether in a program. If a function reaches the end of the block without encountering a return statement, control simply reverts back to the calling portion of the program without returning any information.Let us consider an example of function without returning any information.

```
#include <stdio.h>

void maxi(int, int); /*function declaration*/

main()

{

        int x,y;

        printf(“Enter two integer values”);

        scanf(“%d %d”’ &x,&y);

        maxi(x,y); /*call to function*/

}

void maxi(int x, int y) /*function definition*/
```

```
        {
                int z;
                z=(x>=y)?x:y;
                print("\n\n Maximum value %d",z);
        }
```

This 'maxi' function do not return any value to the calling program, it simply returns the control to the calling programs, even if return statement is not present, then also program will work efficiently.

8.3 ACCESSEMENT OF A FUNCTION

A function can be accessed by specifying its name, followed by a list of parameters or arguments enclosed in parentheses and separated by commas. If the function call does not require any arguments an empty pair of parentheses must follow the function's name. The function call may appear by itself or it may be one of the operands within a more complex expression. The parameters in the body of the functions are called actual arguments as stated earlier; they may be expressed as constants, single variables or more complex expressions. Let us consider another example of function.

```
        #include <stdio.h>
        int sum_v(int,int);
        void main()
        {
                int a,b,c;
                printf("Enter two numbers");
                scanf("%d%d", &a,&b);
                c=sum_v(a,b,);
                printf("\n The sum of two variables is %d\n,",c);
        }
        int sum_v(int a, int b)
        {
                int d;
                d=a+b;
        return d;

    }
```

This program returns the sum of two variables **a** and **b** to the calling program from where sum_v is executing. The sum is present in the variable **c** through the '**return d'** statement. There may be several different calls to the same function from various places within a program. The actual parameters may differ from one function call to another. Within each function call, the actual arguments must correspond to the formal arguments in the function definition, i.e. the number of actual arguments must be same as the number of formal arguments and each actual argument must be of the same data type as its corresponding formal argument. Let us consider an example.

```
    #include <stdio.h>

    maxi(int, int);

    void main()

    {

        int a,b,c,d;

        printf("\n Enter value of a=");

        scanf("%d", &a);

        printf("\n Enter value of b=");

        scanf("%d",&b);

        printf("\n Enter value of c=");

        scanf("%d", &c);

        d=maxi(a,b);

        printf("\n maximum =%d", maxi(c,d));

    }

    maxi(int x,int y);

    {

        int z;

        z=(x>=y)? x:y;

        return z;

    }
```

The function maxi is accessed from two different places in main. In the first call actual rguments are a, b and in the second call c, d are the actual arguments.

If a function returns a non-integer quantity and the portion of the program containing the function call precedes the function definition, then there must be a function declaration in the calling portion of the program. The function declaration effectively informs the compiler that a function will be accessed before it is defined. A function declaration can be written as.

datatype function_ name ( );

Function calls can span several levels within a program; function A can call function B so on.

8.4 PASSING ARGUMENT TO A FUNCTION

Arguments can be passed to a function by two methods, they are called passing by value and passing by reference. When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change. This procedure for passing the value of an argument to a function is known as passing by value. Let us consider an example

```
#include <stdio.h>

void main()

{

        int x=3;

        printf("\n x=%d(from main, before calling the function"),x);

        change(x);

        printf("\n\nx=%d(from main, after calling the function)",x);

}

change(int x)

{

        x=x+3;

        printf("\nx=%d(from the function, after being modified)",x);

}
```

The original value of x (i.e. x=3) is displayed when main begins execution. This value is then passed to the function change, where it is sum up by 3 and the new value displayed. This new value is the altered value of the formal argument that is displayed within the function.

Finally, the value of x within main is again displayed, after control is transferred back to main from change.

x=3 (from main, before calling the function)

x=6 (from the function, after being modified)

x=3 (from main, after calling the function)

Passing an argument by value allows a single-valued actual argument to be written as an expression rather than being restricted to a single variable. But it prevents information from being transferred back to the calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information. Arrays are passed differently than single-valued entities. If an array name is specified as an actual argument, the individual array elements are not copied to the function. Instead the location of the array is passed to the function. If an element of the array is accessed within the function, the access will refer to the location of that array element relative to the location of the first element. Thus, any alteration to an array element within the function will carry over to the calling routine.

8.5 SPECIFICATION OF DATA TYPES OF ARGUMENTS

The calling portion of a program must contain a function declaration if a function returns a non-integer value and the function call precedes the function definition. Function declaration may be included in the calling portion of a program even if it is not necessary. It is possible to include the data types of the arguments within the function declaration. The compiler will then convert the value of each actual argument to the declared data type and then compare each actual data type with its corresponding formal argument. Compilation error will result if the data types do not agree. We had already been use, data types of the arguments within the function declaration. When the argument data types are specified in a function declaration, the general form of the function declaration can be written as:

data-type function name (argument type1, argument type2, … argumenttype n);

Where data-type is the data type of the quantity returned by the function, function name is the name of function, and argument type/refer to the data types of the first argument and so on. Argument data types can be omitted, even if situations require a function declaration. Most C compilers support the use of the keyword void in function definitions, as a return data type indicating that the function does not return anything. Function declarations may also include void for the same purpose. In addition, void may appear in an argument list, in both function definitions and function declarations, to indicate that a function does not require arguments.

8.6 FUNCTION PROTOTYPES AND RECURSION

Many C compilers permits each of the argument data types within a function declaration to be followed by an argument name, that is:

data-type function name (type1 argument 1, type 2 argument2…type n argument n);

Function declarations written in this form are called function prototypes. Function prototypes are desirable, however, because they further facilitate error checking between the calls to a function and the corresponding function definition. Some of the function prototypes are given below:

int example (int, int); or int example (int a, int b);

void example 1(void); or void example 1(void);

void fun (char, long); or void fun (char c, long f);

The names of the arguments within the function declaration need not be declared elsewhere in the program, since these are "dummy" argument names recognized only within the declaration. "C" language also permits the useful feature of 'Recursion'.

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a precious result. In order to solve a problem recursively, two conditions must be satisfied. The problem must be written in a recursive form, and the problem statement must include a stopping condition. The best example of recursion is calculation of factorial of a integer quantity, in which the same procedure is repeating itself. Let us consider the example of factorial:

```
#include <stdio.h>

long int rec(int number);

void main()

{

int no, fact ;

printf ( "\nEnter any number " ) ;

scanf ( "%d", &no ) ;

fact = rec ( no ) ;

printf ( "Factorial value = %d", fact ) ;

}

rec ( int x )

{

int f ;

if ( x == 1 )
```

```
 return ( 1 ) ;

else

f = x * rec ( x - 1 ) ;

return ( f ) ;

}
```

And here is the output for four runs of the program

```
Enter any number 1

Factorial value = 1

Enter any number 2

Factorial value = 2

Enter any number 3

Factorial value = 6

Enter any number 5

Factorial value = 120
```

The point to be noted here is that the function ‘rec’ calls itself recursively, with an actual argument (n-1) that decrease in value for each successive call. The recursive calls terminate the value of the actual argument becomes equal to 1. When a recursive program is executed, the recursive function calls are not executed immediately. Instead of it, they are placed on a stack until the condition that terminates the recursion is encountered. The function calls are then executed in reverse order, as they are popped off the stack. The use of recursion is not necessarily the best way to approach a problem, even though the problem definition may be recursive in nature.

Lesson -9 Library Functions

9.1 INTRODUCTION

C provides a large set of standard library functions for specialized purposes. These may be divided into various categories. For instance

- Character identification (ctype.h)
- Mathematical functions (math.h)
- String manipulation (string.h)

A program generally has to #include special header files in order to use special functions in libraries. The names of the appropriate files can be found in particular compiler manuals. In the examples above the names of the header files are given in parentheses

9.2 Character identification <ctype.h>

The **ctype.h** header file of the C Standard Library provides declares several functions useful for testing and mapping characters.

All the functions accepts **int** as a parameter, whose value must be EOF or representable as an unsigned char.

All the functions return non-zero (true) if the argument c satisfies the condition described, and zero if not.

9.2.1 Library Functions

Following are the functions defined in the header **ctype.h**:

S.N.	Function & Description
1	int isalnum(int c) This function check whether the passed character is alphanumeric.
2	int isalpha(int c) This function check whether the passed character is alphabetic.
3	int iscntrl(int c) This function check whether the passed character is control character.
4	int isdigit(int c) This function check whether the passed character is decimal digit.
5	int isgraph(int c)

	This function check whether the passed character has graphical representation using locale.
6	int islower(int c) This function check whether the passed character is lowercase letter.
7	int isprint(int c) This function check whether the passed character is printable .
8	int ispunct(int c) This function check whether the passed character is punctuation character.
9	int isspace(int c) This function check whether the passed character is white-space.
10	int isupper(int c) This function check whether the passed character is uppercase letter.
11	int isxdigit(int c) This function check whether the passed character is hexadecimal digit.

The library also contains two conversion functions that also accept and return an "int"

S.N.	Function & Description
1	int tolower(int c) This function convert uppercase letter to lowercase.
2	int toupper(int c) This function convert lowercase letter to uppercase.

9.2.2 isalnum()

- **Description:** The C library function **void isalnum(int c)** checks if the passed character is alphanumeric.
- **Syntax:** int isalnum(int c);
- **Parameters:** c -- This is the character to be checked.
- **Return Value:** This function returns nonzero value if c is a digit or a letter, else 0

The following example shows the usage of isalnum() function.

```
#include <stdio.h>

#include <ctype.h>

int main()
```

```
{
            int var1 = 'd';
           int var2 = '\t';
            if( isalnum(var1) )
           {
                        printf("var1 = |%c| is alphanumeric\n", var1 );
           }
           else
           {
                        printf("var1 = |%c| is not alphanumeric\n", var1 );
           }
           if( isalnum(var2) )
           {
                       printf("var2 = |%c| is alphanumeric\n", var2 );
           }
            else
           {
                        printf("var2 = |%c| is not alphanumeric\n", var2 );
           }
   return(0);
}
```

Output:

```
var1 = |d| is alphanumeric
var2 = |\t| is not alphanumeric
```

9.2.3 isalpha ()

- **Description:** The C library function **void isalpha(int c)** checks if the passed character is alphabetic.

- **Syntax:** int isalpha(int c);
- **Parameters:** c -- This is the character to be checked
- **Return Value:** This function returns nonzero value if c is a alphabet, else 0

The following example shows the usage of isalpha() function.

```
#include <stdio.h>

#include <ctype.h>

int main()

{

  int var1 = 'd';

  int var2 = '2';

  if( isalpha(var1) )

  {

    printf("var1 = |%c| is an alphabet\n", var1 );

  }

  else

  {

    printf("var1 = |%c| is not an alphabet\n", var1 );

  }

  if( isalpha(var2) )

  {

    printf("var2 = |%c| is an alphabet\n", var2 );

  }

  else

  {

    printf("var2 = |%c| is not an alphabet\n", var2 );

  }
```

```
 return(0);

}
```

Output:

```
var1 = |d| is an alphabet

var2 = |2| is not an alphabet
```

9.2.4 isdigit()

- **Description:** The C library function **void isdigit(int c)** checks if the passed character is a decimal digit character.Decimal digits are(numbers): 0 1 2 3 4 5 6 7 8 9
- **Syntax:** int isdigit(int c);
- **Parameters:** c -- This is the character to be checked.
- **Return Value:** This This function returns nonzero value if c is a digit, else 0

The following example shows the usage of isdigit() function.

```
#include <stdio.h>

#include <ctype.h>

int main()

{

        int var1 = 'h';

        int var2 = '2';

        if( isdigit(var1) )

        {

                printf("var1 = |%c| is a digit\n", var1 );

        }

        else

        {

                printf("var1 = |%c| is not a digit\n", var1 );

        }

         if( isdigit(var2) )
```

```
          {
                    printf("var2 = |%c| is a digit\n", var2 );
          }
          else
          {
                    printf("var2 = |%c| is not a digit\n", var2 );
           }
  return(0);
}
```

Output:

```
var1 = |h| is not a digit
var2 = |2| is a digit
```

9.2.5 islower()

- **Description:** The C library function **int islower(int c)** check whether the passed character is lowercase letter.
- **Syntax:** int islower(int c);
- **Parameters: c** -- This is the character to be checked.
- **Return Value:** This function returns a non zero value(true) if c is a lowercase alphabetic letter else, zero (false)

The following example shows the usage of islower() function.

```
#include <stdio.h>
#include <ctype.h>
int main()
{
          int var1 = 'Q';
          int var2 = 'q';
```

```
            if( islower(var1) )
           {
                       printf("var1 = |%c| is lowercase character\n", var1 );
           }
            else
           {
                       printf("var1 = |%c| is not lowercase character\n", var1 );
           }
            if( islower(var2) )
           {
                       printf("var2 = |%c| is lowercase character\n", var2 );
           }
           else
           {
           printf("var2 = |%c| is not lowercase character\n", var2 );
           }
  return(0);
}
```

Let us compile and run the above program, this will produce the following result:

```
var1 = |Q| is not lowercase character
var2 = |q| is lowercase character
```

9.2.6 ispunct()

- **Description:** The C library function **int ispunct(int c)** check whether the passed character is punctuation character.A punctuation character is any graphic character (as in isgraph) that is not alphanumeric (as in isalnum).
- **Syntax:** int ispunct(int c);
- **Parameters: c** -- This is the character to be checked.

- **Return Value:** This function returns a non zero value(true) if c is a punctuation character else, zero (false)

The following example shows the usage of ispunct() function.

```
#include <stdio.h>

#include <ctype.h>

int main()

{

         int var1 = 't';

         int var2 = '/';

        if( ispunct(var1) )

        {

                 printf("var1 = |%c| is a punctuation character\n", var1 );

        }

        else

        {

                printf("var1 = |%c| is not a punctuation character\n", var1 );

        }

        if( ispunct(var2) )

        {

                printf("var2 = |%c| is a punctuation character\n", var2 );

        }

        else

        {

                printf("var2 = |%c| is not a punctuation character\n", var2 );

        }

 return(0);

}
```

Output:

```
var1 = |t| is not a punctuation character

var2 = |/| is a punctuation character
```

9.2.7 isspace()

- **Description:**The C library function **int isspace(int c)** check whether the passed character is white-space.
- **Syntax:** int isspace(int c);
- **Parameters: c** -- This is the character to be checked.
- **Return Value:** This function returns a non zero value(true) if c is a white-space character else, zero (false)

The following example shows the usage of isspace() function.

```
#include <stdio.h>

#include <ctype.h>

int main()

{

        int var1 = 't';

        int var2 = ' ';

        if( isspace(var1) )

        {

                printf("var1 = |%c| is a white-space character\n", var1 );

        }

        else

        {

                printf("var1 = |%c| is not a white-space character\n", var1 );

        }

        if( isspace(var2) )

        {
```

```
                    printf("var2 = |%c| is a white-space character\n", var2 );
          }
           else
          {
                    printf("var2 = |%c| is not a white-space character\n", var2 );
          }
  return(0);
}
```

Output:

```
var1 = |t| is not a white-space character
var2 = | | is a white-space character
```

9.2.8 issupper()

- **Description:** The C library function **int isupper(int c)** check whether the passed character is uppercase letter.
- **Declaration:** Following is the declaration for isupper() function.
- **Syntax:** int isupper(int c);
- **Parameters: c** -- This is the character to be checked.
- **Return Value:** This function returns a non zero value(true) if c is uppercase alphabetic letter else, zero (false)

The following example shows the usage of isupper() function.

```
#include <stdio.h>
#include <ctype.h>
int main()
{
          int var1 = 'M';
          int var2 = 'm';
          if( isupper(var1) )
          {
                    printf("var1 = |%c| is uppercase character\n", var1 );
          }
          else
          {
                    printf("var1 = |%c| is not uppercase character\n", var1 );
          }
          if( isupper(var2) )
          {
                    printf("var2 = |%c| is uppercase character\n", var2 );
          }
          else
          {
                    printf("var2 = |%c| is not uppercase character\n", var2 );
          }
  return(0);
}
```

Output:

```
var1 = |M| is uppercase character
var2 = |m| is not uppercase character
```

9.2.9 isxdigit()

- **Description:** The C library function **int isxdigit(int c)** check whether the passed character is hexadecimal digit.
- **Declaration:** Following is the declaration for isxdigit() function.
- Syntax: int isxdigit(int c);
- **Parameters: c** -- This is the character to be checked.

- **Return Value:** This function returns a non zero value(true) if c is hexadecimal digit else, zero (false)

The following example shows the usage of isxdigit() function.

```
#include <stdio.h>

#include <ctype.h>

int main()

{

        char var1[] = "tuts";

        char var2[] = "0xE";

        if( isxdigit(var1[0]) )

        {

                printf("var1 = |%s| is hexadecimal character\n", var1 );

        }

        else

        {

                printf("var1 = |%s| is not hexadecimal character\n", var1 );

        }

        if( isxdigit(var2[0] ))

        {

                printf("var2 = |%s| is hexadecimal character\n", var2 );

        }

        else

        {

                 printf("var2 = |%s| is not hexadecimal character\n", var2 );

        }

  return(0);

}
```

Output:

```
var1 = |tuts| is not hexadecimal character

var2 = |0xE| is hexadecimal character
```

9.2.10 tolower()

- **Description:**The C library function **int tolower(int c)** converts a given letter to lowercase.
- **Declaration:** Following is the declaration for tolower() function.
- **Syntax:** int tolower(int c);
- **Parameters: c** -- This is the letter to be converted to lowercase.
- **Return Value:** This function returns lowercase equivalent to c, if such value exists, else c remain unchanged. The value is returned as an int value that can be implicitly casted to char.

The following example shows the usage of tolower() function.

```
#include <stdio.h>

#include <ctype.h>

int main()

{

  int i = 0;

  char c;

  char str[] = "TUTORIALS POINT";

  while( str[i] )

  {

    putchar(tolower(str[i]));

    i++;

  }

  return(0);

}
```

Output:

tutorials point

9.2.11 toupper()

- **Description:** The C library function **int toupper(int c)** converts lowercase letter to uppercase.
- **Declaration:** Following is the declaration for toupper() function.
- **Syntax:** int toupper(int c);
- **Parameters: c** -- This is the letter to be converted to uppercase.
- **Return Value:** This function returns uppercase equivalent to c, if such value exists, else c remain unchanged. The value is returned as an int value that can be implicitly casted to char.

The following example shows the usage of toupper() function.

```
#include <stdio.h>

#include <ctype.h>

int main()

  int i = 0;

  char c;

  char str[] = "Tutorials Point";

  while(str[i])

  {

    putchar (toupper(str[i]));

    i++;

  }

 return(0);

}
```

Output:

```
TUTORIALS POINT
```

9.3 Mathematical functions <math.h>

The **math.h** header defines various mathematical functions and one macro. All the functions available in this library take **double** as an argument and return **double** as the result.

Following are the functions defined in the header math.h:

S.N.	Function & Description
1	double acos(double x) Returns the arc cosine of x in radians.
2	double asin(double x) Returns the arc sine of x in radians.
3	double atan(double x) Returns the arc tangent of x in radians.
4	double atan2(doubly y, double x) Returns the arc tangent in radians of y/x based on the signs of both values to determine the correct quadrant.
5	double cos(double x) Returns the cosine of a radian angle x.
6	double cosh(double x) Returns the hyperbolic cosine of x.
7	double sin(double x) Returns the sine of a radian angle x.
8	double sinh(double x) Returns the hyperbolic sine of x.
9	double tanh(double x) Returns the hyperbolic tangent of x.
10	double exp(double x) Returns the value of e raised to the xth power.
11	double frexp(double x, int *exponent) The returned value is the mantissa and the integer pointed to by exponent is the exponent. The resultant value is x = mantissa * 2 ^ exponent.
12	double ldexp(double x, int exponent) Returns x multiplied by 2 raised to the power of exponent.
13	double log(double x) Returns the natural logarithm (base-e logarithm) of x.
14	double log10(double x) Returns the common logarithm (base-10 logarithm) of x.

15	double modf(double x, double *integer) The returned value is the fraction component (part after the decimal), and sets integer to the integer component.
16	double pow(double x, double y) Returns x raised to the power of y.
17	double sqrt(double x) Returns the square root of x.
18	double ceil(double x) Returns the smallest integer value greater than or equal to x.
19	double fabs(double x) Returns the absolute value of x
20	double floor(double x) Returns the largest integer value less than or equal to x.
21	double fmod(double x, double y) Returns the remainder of x divided by y.

9.3.1 acos()

- **Description:** The C library function **double acos(double x)** returns the arc cosine of **x** in radians.
- **Declaration:** Following is the declaration for acos() function.
- Syntax: double acos(double x)
- **Parameters: x** -- This is the floating point value in the interval [-1,+1].
- **Return Value:** This function returns principal arc cosine of x, in the interval [0, pi] radians.

The following example shows the usage of acos() function.

```
#include <stdio.h>

#include <math.h>

#define PI 3.14159265

int main ()

{

        double x, ret, val;

        x = 0.9;
        val = 180.0 / PI;

         ret = acos(x) * val;

        printf("The arc cosine of %lf is %lf degrees", x, ret);

        return(0);

}
```

Output:

```
The arc cosine of 0.900000 is 25.855040 degrees
```

9.3.2 asin()

- **Description**: The C library function **double asin(double x)** returns the arc sine of **x** in radians.
- **Declaration:** Following is the declaration for asin() function.
- Syntax: double asin(double x)
- **Parameters: x** -- This is the floating point value in the interval [-1,+1].
- **Return Value:** This function returns the arc sine of x, in the interval [-pi/2,+pi/2] radians.

The following example shows the usage of asin() function.

```
#include <stdio.h>

#include <math.h>

#define PI 3.14159265

int main ()

{

        double x, ret, val;

        x = 0.9;

        val = 180.0 / PI;

        ret = asin(x) * val;

        printf("The arc sine of %lf is %lf degrees", x, ret);

        return(0);
}
```

Let us compile and run the above program, this will produce the following result:

```
The arc sine of 0.900000 is 64.190609 degrees
```

9.3.3 atan()

- **Description:** The C library function **double atan(double x)** returns the arc tangent of **x** in radians.
- **Declaration:** Following is the declaration for atan() function.
- **Syntax:** double atan(double x)
- **Parameters: x** -- This is the floating point value.
- **Return Value:** This function returns the principal arc tangent of x, in the interval [-pi/2,+pi/2] radians.

The following example shows the usage of atan() function.

```
#include <stdio.h>

#include <math.h>

#define PI 3.14159265

int main ()

{

          double x, ret, val;

          x = 1.0;

          val = 180.0 / PI;

            ret = atan (x) * val;

          printf("The arc tangent of %lf is %lf degrees", x, ret);

            return(0);

}
```

Output:

```
The arc tangent of 1.000000 is 45.000000 degrees
```

9.3.4 exp()

- **Description:** The C library function **double exp(double x)** returns the value of **e** raised to the **xth** power.
- **Declaration:** Following is the declaration for exp() function.
- **Syntax:** double exp(double x)
- **Parameters: x** -- This is the floating point value.
- **Return Value:** This function returns the exponential value of x.

The following example shows the usage of exp() function.

```
#include <stdio.h>

#include <math.h>

int main ()

{

            double x = 0;

             printf("The exponential value of %lf is %lf\n", x, exp(x));

            printf("The exponential value of %lf is %lf\n", x+1, exp(x+1));

            printf("The exponential value of %lf is %lf\n", x+2, exp(x+2));

              return(0);

}
```

Output:

```
The exponential value of 0.000000 is 1.000000

The exponential value of 1.000000 is 2.718282

The exponential value of 2.000000 is 7.389056
```

9.3.5 log()

- **Description:** The C library function **double log(double x)** returns the natural logarithm (base-e logarithm) of **x**.
- **Declaration:** Following is the declaration for log() function.
- Syntax: double log(double x)

- **Parameters: x** -- This is the floating point value.
- **Return Value:** This function returns natural logarithm of x.

The following example shows the usage of log() function.

```
#include <stdio.h>

#include <math.h>

int main ()

{

            double x, ret;

             x = 2.7;

            /* finding log(2.7) */

            ret = log(x);

            printf("log(%lf) = %lf", x, ret);

            return(0);

}
```

Output:

```
log(2.700000) = 0.993252
```

9.3.6 log10()

- **Declaration:** Following is the declaration for log10() function.
- **Syntax:** double log10(double x)
- **Parameters: x** -- This is the floating point value.
- **Return Value:** This function returns the common logarithm of x, for values of x greater than zero.

The following example shows the usage of log10() function.

```
#include <stdio.h>

#include <math.h>

int main ()

{
```

```
          double x, ret;

          x = 10000;

          /* finding value of log

10

10000 */

          ret = log10(x);

          printf("log10(%lf) = %lf\n", x, ret);

            return(0);

}
```

Output:

```
log10(10000.000000) = 4.000000
```

9.3.7 pow()

- Description: The C library function **double pow(double x, double y)** returns **x** raised to the power of **y** i.e. x^y.
- Declaration: Following is the declaration for pow() function.
- double pow(double x, double y)
- Parameters: **x** -- This is the floating point base value.

y -- This is the floating point power value.

- Return Value: This function returns the result of raising x to the power y.

The following example shows the usage of pow() function.

```
#include <stdio.h>

#include <math.h>

int main ()

{

          printf("Value 8.0 ^ 3 = %lf\n", pow(8.0, 3));

          printf("Value 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));

    return(0);
}
```

Output:

```
Value 8.0 ^ 3 = 512.000000

Value 3.05 ^ 1.98 = 9.097324
```

9.3.8 sqrt()

- **Description:** The C library function **double sqrt(double x)** returns the square root of **x**.
- **Declaration:** Following is the declaration for sqrt() function.
- **Syntax:** double sqrt(double x)
- **Parameters: x** -- This is the floating point value.
- **Return Value:** This function returns the square root of x.

The following example shows the usage of sqrt() function.

```
#include <stdio.h>

#include <math.h>

int main ()

{

        printf("Square root of %lf is %lf\n", 4.0, sqrt(4.0) );

        printf("Square root of %lf is %lf\n", 5.0, sqrt(5.0) );

        return(0);

}
```

Output:

```
Square root of 4.000000 is 2.000000

Square root of 5.000000 is 2.236068
```

9.3.9 ceil()

- **Description:** The C library function **double ceil(double x)** returns the smallest integer value greater than or equal to **x**.
- **Declaration:** Following is the declaration for ceil() function.
- **Syntax:** double ceil(double x)

- **Parameters: x** -- This is the floating point value.
- **Return Value:** This function returns the smallest integral value not less than x.

The following example shows the usage of ceil() function.

```
#include <stdio.h>

#include <math.h>

int main ()

{

            float val1, val2, val3, val4;

            val1 = 1.6;

            val2 = 1.2;

            val3 = 2.8;

            val4 = 2.3;

            printf ("value1 = %.1lf\n", ceil(val1));

            printf ("value2 = %.1lf\n", ceil(val2));

             printf ("value3 = %.1lf\n", ceil(val3));

            printf ("value4 = %.1lf\n", ceil(val4));

  return(0);

}
```

Output:

```
value1 = 2.0

value2 = 2.0

value3 = 3.0

value4 = 3.0
```

9.3.10 fabs()

- **Description:** The C library function **double fabs(double x)** returns absolute value of **x**.
- **Declaration:** Following is the declaration for fabs() function.

- **Syntax:** double fabs(double x)
- **Parameters: x** -- This is the floating point value.
- **Return Value**: This function returns the absolute value of x.

The following example shows the usage of fabs() function.

```
#include <stdio.h>

#include <math.h>

int main ()

{

        int a, b;

        a = 1234;

        b = -344;

        printf("The absolute value of %d is %lf\n", a, fabs(a));

        printf("The absolute value of %d is %lf\n", b, fabs(b));

        return(0);

}
```

Output:

```
The absolute value of 1234 is 1234.000000

The absolute value of -344 is 344.000000
```

9.3.11 floor()

- Description: The C library function **double floor(double x)** returns the largest integer value less than or equal to **x**.
- Declaration: Following is the declaration for floor() function.
- Syntax: double floor(double x)
- Parameters: **x** -- This is the floating point value.
- Return Value: This function returns the largest integral value not greater than x.

The following example shows the usage of floor() function.

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

```
#include <math.h>

int main ()

{

            float val1, val2, val3, val4;

            val1 = 1.6;

            val2 = 1.2;

            val3 = 2.8;

            val4 = 2.3;

            printf("Value1 = %.1lf\n", floor(val1));

            printf("Value2 = %.1lf\n", floor(val2));

            printf("Value3 = %.1lf\n", floor(val3));

             printf("Value4 = %.1lf\n", floor(val4));

            return(0);

}
```

Output:

```
Value1 = 1.0

Value2 = 1.0

Value3 = 2.0

Value4 = 2.0
```

9.3.12 fmod()

- **Description:** The C library function **double fmod(double x, double y)** returns the remainder of **x** divided by **y**.
- **Declaration:** Following is the declaration for fmod() function.
- **Syntax:** double fmod(double x, double y)
- **Parameters: x** -- This is the floating point value with the division numerator i.e. x.

y -- This is the floating point value with the division denominator i.e. y.

- Return Value: This function returns the remainder of dividing x/y.

The following example shows the usage of fmod() function.

```
#include <stdio.h>

#include <math.h>

int main ()

{

            float a, b;

            int c;

             a = 9.2;

             b = 3.7;

            c = 2;

             printf("Remainder of %f / %d is %lf\n", a, c, fmod(a,c));

            printf("Remainder of %f / %f is %lf\n", a, b, fmod(a,b));

             return(0);

}
```

Output:

```
Remainder of 9.200000 / 2 is 1.200000

Remainder of 9.200000 / 3.700000 is 1.800000
```

9.4 String manipulation <string.h>

The **string .h** header defines one variable type, one macro and various functions for manipulating arrays of characters.

9.4.1 Following is the variable type defined in the header string.h:

S.N.	Variable & Description
1	size_t This is the unsigned integral type and is the result of the sizeof keyword.

9.4.2 Library Macros

Following is the macro defined in the header string.h:

S.N.	Macro & Description
1	NULL This macro is the value of a null pointer constant.

9.4.3 Following are the functions defined in the header string.h:

S.N.	Function & Description
1	void *memchr(const void *str, int c, size_t n) Searches for the first occurrence of the character c (an unsigned char) in the first n bytes of the string pointed to by the argument str.
2	int memcmp(const void *str1, const void *str2, size_t n) Compares the first n bytes of str1 and str2.
3	void *memcpy(void *dest, const void *src, size_t n) Copies n characters from src to dest.
4	void *memmove(void *dest, const void *src, size_t n) Another function to copy n characters from str2 to str1.
5	void *memset(void *str, int c, size_t n) Copies the character c (an unsigned char) to the first n characters of the string pointed to by the argument str.
6	char *strcat(char *dest, const char *src) Appends the string pointed to by src to the end of the string pointed to by dest.
7	char *strncat(char *dest, const char *src, size_t n) Appends the string pointed to by src to the end of the string pointed to by dest up to n characters long.
8	char *strchr(const char *str, int c) Searches for the first occurrence of the character c (an unsigned char) in the string pointed to by the argument str.
9	int strcmp(const char *str1, const char *str2) Compares the string pointed to by str1 to the string pointed to by str2.
10	int strncmp(const char *str1, const char *str2, size_t n)

	Compares at most the first n bytes of str1 and str2.
11	int strcoll(const char *str1, const char *str2) Compares string str1 to str2. The result is dependent on the LC_COLLATE setting of the location.
12	char *strcpy(char *dest, const char *src) Copies the string pointed to by src to dest.
13	char *strncpy(char *dest, const char *src, size_t n) Copies up to n characters from the string pointed to by src to dest.
14	size_t strcspn(const char *str1, const char *str2) Calculates the length of the initial segment of str1 which consists entirely of characters not in str2.
15	char *strerror(int errnum) Searches an internal array for the error number errnum and returns a pointer to an error message string.
16	size_t strlen(const char *str) Computes the length of the string str up to but not including the terminating null character.
17	char *strpbrk(const char *str1, const char *str2) Finds the first character in the string str1 that matches any character specified in str2.
18	char *strrchr(const char *str, int c) Searches for the last occurrence of the character c (an unsigned char) in the string pointed to by the argument str.
19	size_t strspn(const char *str1, const char *str2) calculates the length of the initial segment of str1 which consists entirely of characters in str2.
20	char *strstr(const char *haystack, const char *needle) Finds the first occurrence of the entire string needle (not including the terminating null character) which appears in the string haystack.
21	char *strtok(char *str, const char *delim) Breaks string str into a series of tokens separated by delim.
22	size_t strxfrm(char *dest, const char *src, size_t n) Transforms the first n characters of the string src into corrent locale and place them in the string dest.

9.4.4 strcat()

- **Description:** The C library function **char *strcat(char *dest, const char *src)** appends the string pointed to by **src** to the end of the string pointed to by **dest**.
- **Declaration:** Following is the declaration for strcat() function.
- **Syntax:** char *strcat(char *dest, const char *src)
- **Parameters: dest** -- This is pointer to the destination array, which should contain a C string, and be large enough to contain the concatenated resulting string.

src -- This is the string to be appended. This should not overlap destination.

- **Return Value:** This function return a pointer to the resulting string dest.

The following example shows the usage of strcat() function.

```
#include <stdio.h>

#include <string.h>

int main ()

{

        char src[50], dest[50];

        strcpy(src,  "This is source");

        strcpy(dest, "This is destination");

        strcat(dest, src);

       printf("Final destination string : |%s|", dest);

        return(0);

}
```

Output:

```
Final destination string : |This is destinationThis is source|
```

9.4.5 strncat()

Description: The C library function **char *strncat(char *dest, const char *src, size_ t n)** appends the string pointed to by **src** to the end of the string pointed to by **dest** up to **n** characters long.

Declaration: Following is the declaration for strncat() function.

Syntax: char *strncat(char *dest, const char *src, size_t n)

Parameters: dest -- This is pointer to the destination array, which should contain a C string, and be large enough to contain the concatenated resulting string which includes the additional null-character.

src -- This is the string to be appended.

n -- This is the maximum number of characters to be appended.

Return Value: This function return a pointer to the resulting string dest.

The following example shows the usage of strncat() function.

```
#include <stdio.h>

#include <string.h>

int main ()

{

        char src[50], dest[50];

        strcpy(src,  "This is source");

        strcpy(dest, "This is destination");

        strncat(dest, src, 15);

        printf("Final destination string : |%s|", dest);

        return(0);

}
```

Output:

```
Final destination string : |This is destinationThis is source|
```

9.4.6 strchr()

- **Description:** The C library function **char *strchr(const char *str, int c)** searches for the first occurrence of the character **c** (an unsigned char) in the string pointed to by the argument **str**.
- **Declaration:** Following is the declaration for strchr() function.
- **Syntax:** char *strchr(const char *str, int c)
- **Parameters: str** -- This is the C string to be scanned.

c -- This is the character to be searched in str.

- **Return Value:** This returns a pointer to the first occurrence of the character c in the string str, or NULL if the character is not found.

The following example shows the usage of strchr() function.

```
#include <stdio.h>

#include <string.h>
```

```
int main ()

{

        const char str[] = "http://www.tutorialspoint.com";

        const char ch = '.';

        char *ret;

        ret = strchr(str, ch);

        printf("String after |%c| is - |%s|\n", ch, ret);

        return(0);

}
```

Output:

```
String after |.| is - |.tutorialspoint.com|
```

9.4.7 strcmp()

- Description: The C library function **int strcmp(const char *str1, const char *str2)** compares the string pointed to by **str1** to the string pointed to by **str2**.
- Declaration: Following is the declaration for strcmp() function.
- Syntax: int strcmp(const char *str1, const char *str2)
- Parameters: **str1** -- This is the first string to be compared.

str2 -- This is the second string to be compared.

- Return Value: This function returned values are as follows:
- if Return value if < 0 then it indicates str1 is less than str2
- if Return value if > 0 then it indicates str2 is less than str1
- if Return value if = 0 then it indicates str1 is equal to str2

The following example shows the usage of strncmp() function.

```
#include <stdio.h>

#include <string.h>

int main ()

{
        char str1[15];

        char str2[15];

        int ret;

        strcpy(str1, "abcdef");

         strcpy(str2, "ABCDEF");

        ret = strcmp(str1, str2);

         if(ret > 0)

        {

                printf("str1 is less than str2");

        }

        else if(ret < 0)

        {

                printf("str2 is less than str1");

        }

        else

        {

                printf("str1 is equal to str2");

        }

        return(0);

}
```

Output:

```
str1 is less than str2
```

9.4.8 strcpy()

- **Description:** The C library function **char *strcpy(char *dest, const char *src)** copies the string pointed to by **src** to **dest**.
- **Declaration:** Following is the declaration for strcpy() function.
- **Syntax:** char *strcpy(char *dest, const char *src)

- **Parameters:**
    - **dest** -- This is the pointer to the destination array where the content is to be copied.
    - **src** -- This is the string to be copied.
- Return Value: This return a pointer to the destination string dest.

Output:

```
#include <stdio.h>

#include <string.h>

int main()

{

        char src[40];

        char dest[12];

        memset(dest, '\0', sizeof(dest));

        strcpy(src, "This is tutorialspoint.com");

        strcpy(dest, src);

        printf("Final copied string : %s\n", dest);

        return(0);

}
```

Output:

```
Final copied string : This is tutorialspoint.com
```

9.4.9 strcspn()

- **Description:** The C library function **size_t strcspn(const char *str1, const char *str2)** calculates the length of the initial segment of **str1** which consists entirely of characters not in **str2**.
- **Declaration:** Following is the declaration for strcspn() function.
- **Syntax:** size_t strcspn(const char *str1, const char *str2)
- **Parameters:**
    - **str1** -- This is the main C string to be scanned.

- **str2** -- This is the string containing a list of characters to match in str1.

- **Return Value:** This function returns the number of characters in the initial segment of string str1 which are not in the string str2.

The following example shows the usage of strcspn() function.

```
#include <stdio.h>

#include <string.h>

int main ()

{

        int len;

        const char str1[] = "ABCDEF4960910";

        const char str2[] = "013";

        len = strcspn(str1, str2);

        printf("First matched character is at %d\n", len + 1);

        return(0);

}
```

Output:

```
First matched character is at 10
```

9.4.10 strlen()

- **Description:** The C library function **size_t strlen(const char *str)** computes the length of the string **str** up to but not including the terminating null character.
- **Declaration:** Following is the declaration for strlen() function.
- **Syntax:** size_t strlen(const char *str)
- **Parameters: str** -- This is the string whose length is to be found.
- **Return Value:** This function returns the length of string.

The following example shows the usage of strlen() function.

```
#include <stdio.h>

#include <string.h>
```

```
int main ()
{
  char str[50];
  int len;
  strcpy(str, "This is tutorialspoint.com");
  len = strlen(str);
  printf("Length of |%s| is |%d|\n", str, len);
    return(0);
}
```

Output:

```
Length of |This is tutorialspoint.com| is |26|
```

9.4.11 strrchr()

- **Description:** The C library function **char *strrchr(const char *str, int c)** searches for the last occurrence of the character **c** (an unsigned char) in the string pointed to by the argument **str**.
- **Declaration:** Following is the declaration for strrchr() function.
- **Syntax:** char *strrchr(const char *str, int c)
- **Parameters:**

str -- This is the C string.

c -- This is the character to be located. It is passed as its int promotion, but it is internally converted back to char.

Return Value: This function returns a pointer to the last occurrence of character in str. If the value is not found, the function returns a null pointer.

The following example shows the usage of strrchr() function.

```
#include <stdio.h>
#include <string.h>
int main ()
{
        int len;

        const char str[] = "http://www.tutorialspoint.com";

        const char ch = '.';

         char *ret;

        ret = strrchr(str, ch);

        printf("String after |%c| is - |%s|\n", ch, ret);

        return(0);

}
```

Output:

```
String after |.| is - |.com|
```

9.4.12 strstr()

- **Description:** The C library function **char *strstr(const char *haystack, const char *needle)** function finds the first occurrence of the substring **needle** in the string **haystack**. The terminating '\0' characters are not compared.
- **Declaration:** Following is the declaration for strstr() function.
- **Syntax:** char *strstr(const char *haystack, const char *needle)
- **Parameters:**

haystack -- This is the main C string to be scanned.

needle -- This is the small string to be searched with-in haystack string.

- **Return Value:** This function returns a pointer to the first occurrence in haystack of any of the entire sequence of characters specified in needle, or a null pointer if the sequence is not present in haystack.

The following example shows the usage of strstr() function.

```
#include <stdio.h>

#include <string.h>

int main()

{

        const char haystack[20] = "TutorialsPoint";
        const char needle[10] = "Point";

        char *ret;

        ret = strstr(haystack, needle);

        printf("The substring is: %s\n", ret);

        return(0);

}
```

Output:

```
The substring is: Point
```

Lession – 10 Scope and Visibility of a variable

10.1 Introduction

We have already said all that needs to be said about constants, but we are not finished with variables. To fully define a variable one needs to mention not only its 'type' but also its 'storage class'. In other words, not only do all variables have a data type, they also have a 'storage class'.

We have not mentioned storage classes yet, though we have written several programs in C. We were able to get away with this because storage classes have defaults. If we don't specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used. Thus, variables have certain default storage classes.

From C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept— Memory and CPU registers. It is the variable's storage class that determines in which of these two locations the value is stored.

Moreover, a variable's storage class tells us:

- Where the variable would be stored.
- What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value).
- What is the scope of the variable; i.e. in which functions the value of the variable would be available.
- What is the life of the variable; i.e. how long would the variable exist.

There are four storage classes in C:

- Automatic storage class
- Register storage class
- Static storage class
- External storage class

Let us examine these storage classes one by one.

10.2 Automatic Storage Class

The features of a variable defined to have an automatic storage class are as under:

- Storage: Memory
- Scope: Local to the block in which the variable is defined.
- Life: Till the control remains within the block in which the variable is defined.
- Default initial value: An unpredictable value, which is often called a garbage value.

Following program shows how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

```
void main

{

auto int i, j ;

printf ( "\n%d %d", i, j ) ;

}
```

The output of the above program could be...

```
1211 221
```

where, 1211 and 221 are garbage values of **i** and **j**. When you run this program you may get different values, since garbage values are unpredictable. So always make it a point that you initialize the automatic variables properly, otherwise you are likely to get unexpected results. Note that the keyword for this storage class is **auto**, and not automatic.

Scope and life of an automatic variable is illustrated in the following program.

```
void main()

{

auto int i = 1 ;

{

{

printf ( "\n%d ", i ) ;

}

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

The output of the above program is:

```
1 1 1
```

This is because, all **printf( )** statements occur within the outermost block (a block is all statements enclosed within a pair of braces) in which **i** has been defined. It means the scope of **i** is local to the block in which it is defined. The moment the control comes out of the block in which the variable is defined, the variable and its value is irretrievably lost. To catch my point, go through the following program.

```
void main()
{
auto int i = 1 ;
{
auto int i = 2 ;
{
auto int i = 3 ;
printf ( "\n%d ", i ) ;
}
printf ( "%d ", i ) ;
}
printf ( "%d", i ) ;
}
```

The output of the above program would be:

```
3 2 1
```

Note that the Compiler treats the three **i**'s as totally different variables, since they are defined in different blocks. Once the control comes out of the innermost block the variable **i** with value 3 is lost, and hence the **i** in the second **printf( )** refers to **i** with value 2. Similarly, when the control comes out of the next innermost block, the third **printf( )** refers to the **i** with value 1.

Understand the concept of life and scope of an automatic storage class variable thoroughly before proceeding with the next storage class.

10.3 Register Storage Class

The features of a variable defined to be of **register** storage class are as under:

- Storage: CPU registers
- Default initial value: Garbage value.
- Scope: Local to the block in which the variable is defined.
- Life: Till the control remains within the block in which the variable is defined.

A value stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as **register**. A good example of frequently used variables is loop counters. We can name their storage class as **register**.

```
main( )

{

register int i ;

for ( i = 1 ; i <= 10 ; i++ )

printf ( "\n%d", i ) ;

}
```

Here, even though we have declared the storage class of **i** as **register**, we cannot say for sure that the value of **i** would be stored in a CPU register. Why? Because the number of CPU registers are limited, and they may be busy doing some other task. What happens in such an event... the variable works as if its storage class is **auto**.

Not every type of variable can be stored in a CPU register.

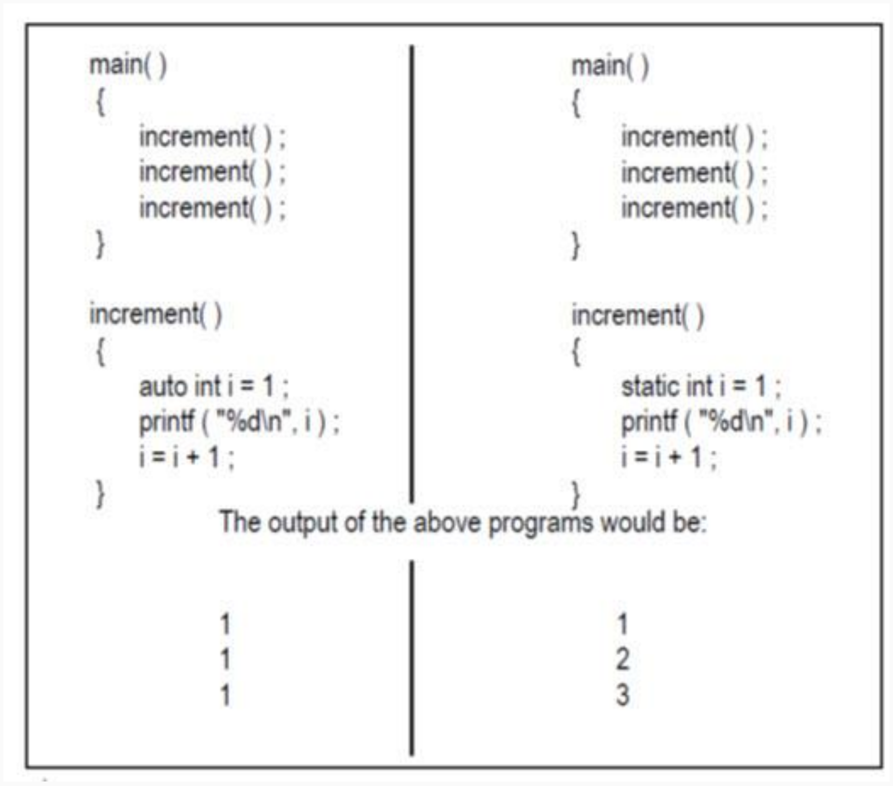
For example, if the microprocessor has 16-bit registers then they cannot hold a **float** value or a **double** value, which require 4 and 8 bytes respectively. However, if you use the **register** storage class for a **float** or a **double** variable you won't get any error messages. All that would happen is the compiler would treat the variables to be of **auto** storage class.

10.4 Static Storage Class

The features of a variable defined to have a **static** storage class are as under:

- Storage: Memory.
- Default initial value: Zero.
- Scope: Local to the block in which the variable is defined.
- Life: Value of the variable persists between different function calls .

Compare the two programs and their output given in following Figure to understand the difference between the **automatic** and **static** storage classes.



```
main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
}

increment( )
{
    auto int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

```
main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
}

increment( )
{
    static int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

The output of the above programs would be:

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

The programs above consist of two functions **main( )** and **increment( )**. The function **increment( )** gets called from **main( )** thrice. Each time it increments the value of **i** and prints it. The only difference in the two programs is that one uses an **auto** storage class for variable **i**, whereas the other uses **static** storage class.

Like **auto** variables, **static** variables are also local to the block in which they are declared. The difference between them is that **static** variables don't disappear when the function is no longer active. Their values persist. If the control comes back to the same function again the **static** variables have the same values they had last time around.

In the above example, when variable **i** is **auto**, each time **increment( )** is called it is re-initialized to one. When the function terminates, **i** vanishes and its new value of 2 is lost. The result: no matter how many times we call **increment( ), i** is initialized to 1 every time.

On the other hand, if **i** is **static**, it is initialized to 1 only once. It is never initialized again. During the first call to **increment( )**, **i** is incremented to 2. Because **i** is static, this value persists. The next time **increment( )** is called, **i** is not re-initialized to 1; on the contrary its old value 2 is still available. This current value of **i** (i.e. 2) gets printed and then **i = i + 1** adds 1 to **i** to get a value of 3. When **increment( )** is called the third time, the current value of **i** (i.e. 3) gets printed and once again **i** is incremented. In short, if the storage class is **static** then the statement **static int i = 1** is executed only once, irrespective of how many times the same function is called.

10.5 External Storage Class

The features of a variable whose storage class has been defined as external are as follows:

- Storage: Memory.
- Default initial value: Zero.
- Scope: Global.
- Life: As long as the program's execution doesn't come to an end.

External variables differ from those we have already discussed in that their scope is global, not local. External variables are declared outside all functions, yet are available to all functions that care to use them. Here is an example to illustrate this fact.

```
int i ;

void main( )

{

printf ( "\ni = %d", i ) ;

increment( ) ;

increment( ) ;

decrement( ) ;

decrement( ) ;

}

increment( )

{

i = i + 1 ;

printf ( "\non incrementing i = %d", i ) ;

}

decrement( )
```

```
{

i = i - 1 ;

printf ( "\non decrementing i = %d", i ) ;

}
```

The output would be:

```
i = 0

on incrementing i = 1

on incrementing i = 2

on decrementing i = 1

on decrementing i = 0
```

As is obvious from the above output, the value of **i** is available to the functions **increment( )** and **decrement( )** since **i** has been declared outside all functions.

Look at the following program.

```
#include<stdio.h>

int x = 21 ;

void main( )

{

extern int y ;

printf ( "\n%d %d", x, y ) ;

}

int y = 31 ;
```

Here, **x** and **y** both are global variables. Since both of them have been defined outside all the functions both enjoy external storage class. Note the difference between the following:

```
extern int y ;

int y = 31 ;
```

Here the first statement is a declaration, whereas the second is the definition. When we declare a variable no space is reserved for it, whereas, when we define it space gets reserved for it in memory. We had to declare **y** since it is being used in **printf( )** before it's definition is encountered. There was no need to declare **x** since its definition is done before its usage. Also remember that a variable can be declared several times but can be defined only once.

Another small issue—what will be the output of the following program?

```
#include<stdio.h>

int x = 10 ;

void main( )

{

int x = 20 ;

printf ( "\n%d", x ) ;

display( ) ;

}

display( )

{

printf ( "\n%d", x ) ;

}
```

Here **x** is defined at two places, once outside **main( )** and once inside it. When the control reaches the **printf( )** in **main( )** which **x** gets printed? Whenever such a conflict arises, it's the local variable that gets preference over the global variable. Hence the **printf( )** outputs 20. When **display( )** is called and control reaches the **printf( )** there is no such conflict. Hence this time the value of the global **x,** i.e. 10 gets printed.

One last thing—a **static** variable can also be declared outside all the functions. For all practical purposes it will be treated as an **extern** variable. However, the scope of this variable is limited to the same file in which it is declared. This means that the variable would not be available to any function that is defined in a file other than the file in which the variable is defined.

10.6 Which to Use When

Dennis Ritchie has made available to the C programmer a number of storage classes with varying features, believing that the programmer is in a best position to decide which one of these storage classes is to be used when. We can make a few ground rules for usage of different storage classes in different programming situations with a view to:

- economise the memory space consumed by the variables
- improve the speed of execution of the program

The rules are as under:

- Use **static** storage class only if you want the value of a variable to persist between different function calls.
- Use **register** storage class for only those variables that are being used very often in a program. Reason is, there are very few CPU registers at our disposal and many of them might be busy doing something else. Make careful utilization of the scarce resources. A typical application of **register** storage class is loop counters, which get used a number of times in a program.
- Use **extern** storage class for only those variables that are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. Declaring all the variables as **extern** would amount to a lot of wastage of memory space because these variables would remain active throughout the life of the program.
- If you don't have any of the express needs mentioned above, then use the **auto** storage class. In fact most of the times we end up using the **auto** variables, because often it so happens that once we have used the variables in a function we don't mind loosing them.

Module 3 – Array, Structure and Pointers

Lesson-11 Array

11.1 INTRODUCTION

An array is a collection of data storage locations, each having the same data type and the same name. Each storage location in an array is called an array element. Why do you need arrays in your programs? let us consider the following program:

```
void main( )

{

                int x ;

                x = 5 ;

x = 10 ;

printf ( "\nx = %d", x ) ;

        }
```

No doubt, this program will print the value of **x** as 10. Why so? Because when a value 10 is assigned to **x**, the earlier value of **x**, i.e. 5, is lost. Thus, ordinary variables (the ones which we have used so far) are capable of holding only one value at a time (as in the above example). However, there are situations in which we would want to store more than one value at a time in a single variable.

For example, suppose we wish to arrange the percentage marks obtained by 100 students in ascending order. In such a case we have two options to store these marks in memory:

- Construct 100 variables to store percentage marks obtained by 100 different students, i.e. each variable containing one student's marks.
- Construct one variable (called array or subscripted variable) capable of storing or holding all the hundred values.

Obviously, the second alternative is better. A simple reason for this is, it would be much easier to handle one variable than handling 100 different variables. Moreover, there are certain logics that cannot be dealt with, without the use of an array. Now a formal definition of an array—An array is a collective name given to a group of 'similar quantities'. These similar quantities could be percentage marks of 100 students, or salaries of 300 employees, or ages of 50 employees. What is important is that the quantities must be 'similar'. Each member in the group is referred to by its position in the group. For example, assume the following group of numbers, which represent percentage marks obtained by five students.

per = { 48, 88, 34, 23, 96 }

If we want to refer to the second number of the group, the usual notation used is per_2. Similarly, the fourth number of the group is referred as per_4. However, in C, the fourth number is referred as **per[3].** This is because in C the counting of elements begins with 0 and not with 1. Thus, in this example **per[3]** refers to 23 and **per[4]** refers to 96. In general, the notation would be **per[i]**, where, **i** can take a value 0, 1, 2, 3, or 4, depending on the position of the element being referred. Here **per** is the subscripted variable (array), whereas **i** is its subscript.

Thus, an array is a collection of similar elements. These similar elements could be all **int**s, or all **float**s, or all **char**s, etc. Usually, the array of characters is called a 'string', whereas an array of **int**s or **float**s is called simply an array. Remember that all elements of any given array must be of the same type. i.e. we cannot have an array of 10 numbers, of which 5 are **int**s and 5 are **float**s.

Arrays can be classified into two different categories:

- Single dimensional arrays
- Multi dimensional arrays

11.2 A Program Using Single Dimensional Array

A single-dimensional array has only a single subscript. A subscript is a number in brackets that follows an array's name. This number can identify the number of individual elements in the array.Let us try to write a program to find average marks obtained by a class of 30 students in a test.

```
void main( )
        {
int avg, sum = 0 ;
int i ;
 int marks[30] ; /* array declaration */
for ( i = 0 ; i <= 29 ; i++ )
{
        printf ( "\nEnter marks " ) ;
        scanf ( "%d", &marks[i] ) ; /* store data in array */
}
for ( i = 0 ; i <= 29 ; i++ )
{
```

```
                              sum = sum + marks[i] ; /* read data from an array*/
                    }
avg = sum / 30 ;
 printf ( "\nAverage marks = %d", avg ) ;
          }
```

There is a lot of new material in this program, so let us take it apart slowly.

11.2.1 Array Declaration

To begin with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want. In our program we have done this with the statement:

```
int marks[30] ;
```

Here, **int** specifies the type of the variable, just as it does with ordinary variables and the word **marks** specifies the name of the variable. The **[30]** however is new. The number 30 tells how many elements of the type **int** will be in our array. The bracket ([ ]) tells the compiler that we are dealing with a single dimensional array.

Accessing Elements of an Array

Once an array is declared, let us see how individual elements in the array can be referred. This is done with subscript, the number in the brackets following the array name. This number specifies the element's position in the array. All the array elements are numbered, starting with 0. Thus, **marks[2]** is not the second element of the array, but the third. In our program we are using the variable **i** as a subscript to refer to various elements of the array. This variable can take different values and hence can refer to the different elements in the array in turn. This ability to use variables as subscripts is what makes arrays so useful.

11.2.2 Entering Data into an Array

Here is the section of code that places data into an array:

```
for ( i = 0 ; i <= 29 ; i++ )
{
printf ( "\nEnter marks " ) ;
scanf ( "%d", &marks[i] ) ;
}
```

The **for** loop causes the process of asking for and receiving a student's marks from the user to be repeated 30 times. The first time through the loop, **i** has a value 0, so the **scanf( )** function will cause the value typed to be stored in the array element **marks[0],** the first element of the array. This process will be repeated until **i** becomes 29. This is last time through the loop, which is a good thing, because there is no array element like **marks[30]**.

In **scanf( )** function, we have used the "address of" operator (&) on the element **marks[i]** of the array, just as we have used it earlier on other variables **(&rate**, for example). In so doing, we are passing the address of this particular array element to the **scanf( )** function, rather than its value; which is what **scanf( )** requires.

11.2.3 Reading Data from an Array

The balance of the program reads the data back out of the array and uses it to calculate the average. The **for** loop is much the same, but now the body of the loop causes each student's marks to be added to a running total stored in a variable called **sum**. When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

```
for ( i = 0 ; i <= 29 ; i++ )

{

sum = sum + marks[i] ;

}

avg = sum / 30 ;

        printf ( "\nAverage marks = %d", avg ) ;
```

11.2.4 More on Arrays

So far we have used arrays that did not have any values in them to begin with. We managed to Array is a very popular data type with C programmers. This is because of the convenience with which arrays lend themselves to programming. The features which make arrays so convenient to program would be discussed below, along with the possible pitfalls in using them.

Array Initialisation store values in them during program execution. Let us now see how to initialize an array while declaring it. Following are a few examples that demonstrate this.

```
int num[6] = { 2, 4, 12, 5, 45, 5 } ;

int n[ ] = { 2, 4, 12, 5, 45, 5 } ;

float press[ ] = { 12.3, 34.2 -23.4, -11.3 } ;
```

Note the following points carefully:

- Till the array elements are not given any specific values, they are supposed to contain garbage values.

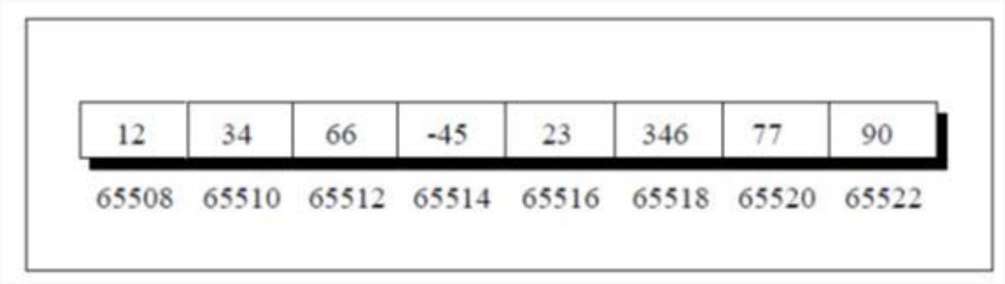
- If the array is initialised where it is declared, mentioning the dimension of the array is optional as in the 2nd example above.

11.2.5 Array Elements in Memory

Consider the following array declaration:

int arr[8] ;

What happens in memory when we make this declaration? 16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers (under Windows/Linux the array would occupy 32 bytes as each integer would occupy 4 bytes). And since the array is not being initialized, all eight values present in it would be garbage values. This so happens because the storage class of this array is assumed to be **auto**. If the storage class is declared to be **static** then all the array elements would have a default initial value as zero. Whatever be the initial values, all the array elements would always be present in contiguous memory locations. This arrangement of array elements in memory is shown in Figure



11.2.6 Bounds Checking

In C there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array; probably on top of other data, or on the program itself. This will lead to unpredictable results, to say the least, and there will be no error message to warn you that you are going beyond the array size. In some cases the computer may just hang. Thus, the following program may turn out to be suicidal.

```
 void main( )

{

 int num[40], i ;

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

        num[i] = i ;

}
```

Thus, to see to it that we do not reach beyond the array size is entirely the programmer's botheration and not the compiler's.

11.3 Two Dimensional Arrays

So far we have explored arrays with only one dimension. It is also possible for arrays to have two or more dimensions. The two-dimensional array is also called a matrix.

Here is a sample program that stores roll number and marks obtained by a student side by side in a matrix.

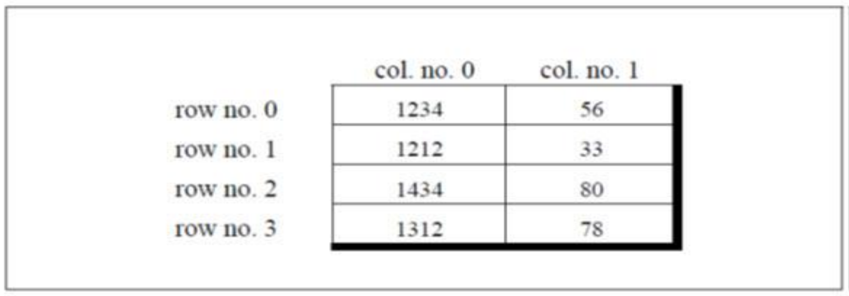
```
          void main( )

          {

                    int stud[4][2] ; int i, j ;

for ( i = 0 ; i <= 3 ; i++ )

                    {

 printf ( "\n Enter roll no. and marks" ) ;

scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;

}

for ( i = 0 ; i <= 3 ; i++ )

{

                              printf ( "\n%d %d", stud[i][0], stud[i][1] ) ;

                    }
```

There are two parts to the program—in the first part through a **for** loop we read in the values of roll no. and marks, whereas, in second part through another **for** loop we print out these values.

Look at the **scanf( )** statement used in the first **for** loop:

```
scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;
```

In **stud[i][0]** and **stud[i][1]** the first subscript of the variable **stud**, is row number which changes for every student. The second subscript tells which of the two columns are we talking about—the zeroth column which contains the roll no. or the first column which contains the marks. Remember the counting of rows and columns begin with zero. The complete array arrangement is shown below.



	col. no. 0	col. no. 1
row no. 0	1234	56
row no. 1	1212	33
row no. 2	1434	80
row no. 3	1312	78

11.3.1 Initializing a 2-Dimensional Array

How do we initialize a two-dimensional array? As simple as this...

```
 int stud[4][2] = { { 1234, 56 }, { 1212, 33 }, { 1434, 80 }, { 1312, 78 } } ;
```

or even this would work...

```
int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
```

of course with a corresponding loss in readability.

It is important to remember that while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Thus the declarations,

```
int arr[2][3] = { 12, 34, 23, 45, 56, 45 } ;

int arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;
```

are perfectly acceptable,

whereas,

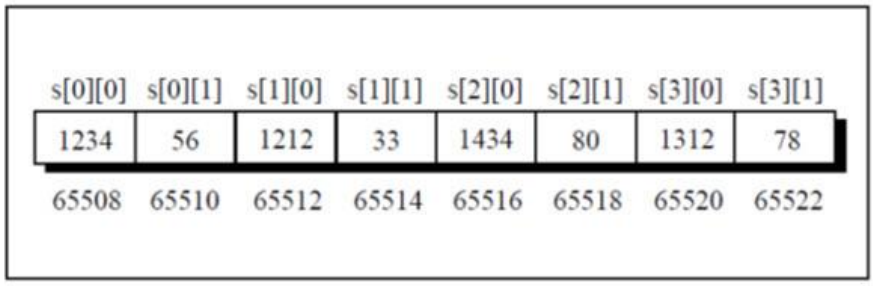
```
int arr[2][ ] = { 12, 34, 23, 45, 56, 45 } ;

 int arr[ ][ ] = { 12, 34, 23, 45, 56, 45 } ;
```

would never work.

11.3.2 Memory Map of a 2-Dimensional Array

Let us reiterate the arrangement of array elements in a two-dimensional array of students, which contains roll nos. in one column and the marks in the other.

The array arrangement shown in Figure 8.4 is only conceptually true. This is because memory doesn’t contain rows and columns. In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain. The arrangement of array elements of a two-dimensional array in memory is shown below:



We can easily refer to the marks obtained by the third student using the subscript notation as shown below:

printf ("Marks of third student = %d", stud[2][1]) ;

Can we not refer the same element using pointer notation, the way we did in one-dimensional arrays? Answer is yes. Only the procedure is slightly difficult to understand. So, read on...

Lesson-12 Structure and Union

12.1 Introduction

Which mechanic is good enough who knows how to repair only one type of vehicle? None. Same thing is true about C language. It wouldn't have been so popular had it been able to handle only all ints, or all floats or all chars at a time. In fact when we handle real world data, we don't usually deal with little atoms of information by themselves—things like integers, characters and such. Instead we deal with entities that are collections of things, each thing having its own attributes, just as the entity we call a 'book' is a collection of things such as title, author, call number, publisher, number of pages, date of publication, etc. As you can see all this data is dissimilar, for example author is a string, whereas number of pages is an integer. For dealing with such collections, C provides a data type called 'structure'. A structure gathers together, different atoms of information that comprise a given entity. And structure is the topic of this chapter.

12.2 Why Structure?

We have seen earlier how ordinary variables can hold one piece of information and how arrays can hold a number of pieces of information of the same data type. These two data types can handle a great variety of situations. But quite often we deal with entities that are collection of dissimilar data types.

For example, suppose you want to store data about a book. You might want to store its name (a string), its price (a float) and number of pages in it (an int). If data about say 3 such books is to be stored, then we can follow two approaches:

- Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.
- Use a structure variable.

Let us examine these two approaches one by one. For the sake of programming convenience assume that the names of books would be single character long. Let us begin with a program that uses arrays.

```
#include<stdio.h>

void main()

{

char name[3] ;

float price[3] ;

int pages[3], i ;
printf ( "\nEnter names, prices and no. of pages of 3 books\n" ) ;

for ( i = 0 ; i <= 2 ; i++ )

scanf ( "%c %f %d", &name[i], &price[i], &pages[i] );

printf ( "\nAnd this is what you entered\n" ) ;

for ( i = 0 ; i <= 2 ; i++ )

printf ( "%c %f %d\n", name[i], price[i], pages[i] );

}
```

And here is the sample run...

```
Enter names, prices and no. of pages of 3 books

A 100.00 354

C 256.50 682

F 233.70 512

And this is what you entered

A 100.000000 354

C 256.500000 682

F 233.700000 512
```

This approach no doubt allows you to store names, prices and number of pages. But as you must have realized, it is an unwieldy approach that obscures the fact that you are dealing with a group of characteristics related to a single entity—the book.

The program becomes more difficult to handle as the number of items relating to the book go on increasing. For example, we would be required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book, etc. To solve this problem, C provides a special data type—the structure.

A structure contains a number of data types grouped together. These data types may or may not be of the same type. The following example illustrates the use of this data type.

```
#include<stdio.h>

void main()

{

        struct book
        {
                char name ;
float price ;
int pages ;
        };
        struct book b1,b2,b3;
printf ( "\nEnter names, prices & no. of pages of 3 books\n" ) ;
scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;
scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;
scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;
printf ( "\nAnd this is what you entered" ) ;
printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;
printf ( "\n%c %f %d", b2.name, b2.price, b2.pages ) ;
printf ( "\n%c %f %d", b3.name, b3.price, b3.pages ) ;
}
```

And here is the output...

```
Enter names, prices and no. of pages of 3 books
A 100.00 354
C 256.50 682
F 233.70 512
A 100.000000 354
C 256.500000 682
F 233.700000 512
```

This program demonstrates two fundamental aspects of structures:

- declaration of a structure
- accessing of structure elements

Let us now look at these concepts one by one.

12.3 Declaring a Structure

In our example program, the following statement declares the structure type:

```
struct book
{
char name ;
float price ;
int pages ;
} ;
```

This statement defines a new data type called **struct book**. Each variable of this data type will consist of a character variable called **name**, a float variable called **price** and an integer variable called **pages**. The general form of a structure declaration statement is given below:

```
struct <structure name>
{
structure element 1 ;
structure element 2 ;
structure element 3 ;
......
......
} ;
```

Once the new structure data type has been defined one or more variables can be declared to be of that type. For example the variables **b1**, **b2**, **b3** can be declared to be of the type **struct book**, as,

```
struct book b1, b2, b3 ;
```

This statement sets aside space in memory. It makes available space to hold all the elements in the structure—in this case, 7 bytes—one for **name**, four for **price** and two for **pages**. These bytes are always in adjacent memory locations.

If we so desire, we can combine the declaration of the structure type and the structure variables in one statement.

For example,

```
struct book

{

char name ;

float price ;

int pages ;

} ;

struct book b1, b2, b3 ;
```

is same as...

```
struct book

{

char name ;

float price ;

int pages ;

} b1, b2, b3 ;
```

or even...

```
struct

{

char name ;

float price ;

int pages ;

} b1, b2, b3 ;
```

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initiate arrays.

```
struct book

{

char name[10] ;

float price ;
int pages ;

} ;

struct book b1 = { "Basic", 130.00, 550 } ;

struct book b2 = { "Physics", 150.80, 800 } ;
```

Note the following points while declaring a structure type:

- The closing brace in the structure type declaration must be followed by a semicolon.
- It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the 'form' of the structure.
- Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive #include) in whichever program we want to use this structure type.

12.4 Accessing Structure Elements

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed.

In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. So to refer to **pages** of the structure defined in our sample program we have to use,

```
b1.pages
```

Similarly, to refer to **price** we would use,

```
b1.price
```

Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

12.5 How Structure Elements are Stored

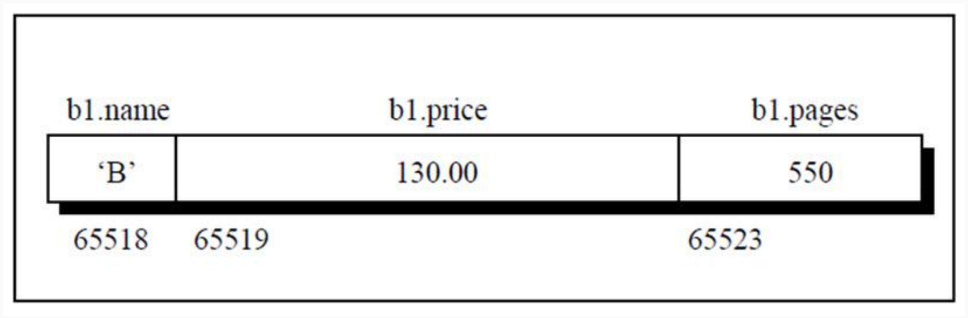
Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```
/* Memory map of structure elements */

void main( )

{

struct book
{
char name ;
float price ;
int pages ;
} ;
struct book b1 = { 'B', 130.00, 550 } ;
printf ( "\nAddress of name = %u", &b1.name ) ;
printf ( "\nAddress of price = %u", &b1.price ) ;
printf ( "\nAddress of pages = %u", &b1.pages ) ;
}
```

Here is the output of the program...

```
Address of name = 65518
Address of price = 65519
Address of pages = 65523
```

Actually the structure elements are stored in memory as shown in the Figure



12.6 Array of Structures

Our sample program showing usage of structure is rather simple minded. All it does is, it receives values into various structure elements and output these values. But that's all we intended to do anyway... show how structure types are created, how structure variables are declared and how individual elements of a structure variable are referenced.

In our sample program, to store data of 100 books we would be required to use 100 different structure variables from **b1** to **b100**, which is definitely not very convenient. A better approach would be to use an array of structures. Following program shows how to use an array of structures.

```
/* Usage of an array of structures */

void main( )

{

struct book

{

char name ;

float price ;

int pages ;

} ;

struct book b[100] ;

int i ;

for ( i = 0 ; i <= 99 ; i++ )

{

printf ( "\nEnter name, price and pages " ) ;

scanf ( "%c %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;

}

for ( i = 0 ; i <= 99 ; i++ )

printf ( "\n%c %f %d", b[i].name, b[i].price, b[i].pages ) ;

}

linkfloat( )

{

float a = 0, *b ;

b = &a ; /* cause emulator to be linked */

a = *b ; /* suppress the warning - variable not used */

}
```

Now a few comments about the program:

- Notice how the array of structures is declared...

struct book b[100] ;

This provides space in memory for 100 structures of the type **struct book**.

The syntax we use to reference each element of the array **b** is similar to the syntax used for arrays of **int**s and **char**s. For example, we refer to zeroth book's price as **b[0].price**. Similarly, we refer first book's pages as **b[1].pages**.

It should be appreciated what careful thought Dennis Ritchie has put into C language. He first defined array as a collection of similar elements; then realized that dissimilar data types that are often found in real life cannot be handled using arrays, therefore created a new data type called structure. But even using structures programming convenience could not be achieved, because a lot of variables (**b1** to **b100** for storing data about hundred books) needed to be handled. Therefore he allowed us to create an array of structures; an array of similar data types which themselves are a collection of dissimilar data types. Hats off to the genius!

In an array of structures all elements of the array are stored in adjacent memory locations. Since each element of this array is a structure, and since all structure elements are always stored in adjacent locations you can very well visualise the arrangement of array of structures in memory. In our example, **b[0]**'s **name**, **price** and **pages** in memory would be immediately followed by **b[1]**'s **name**, **price** and **pages**, and so on.

What is the function **linkfloat( )** doing here? If you don't define it you are bound to get the error "Floating Point Formats Not Linked" with majority of C Compilers. What causes this error to occur? When parsing our source file, if the compiler encounters a reference to the address of a float, it sets a flag to have the linker link in the floating-point emulator. A floating point emulator is used to manipulate floating point numbers in runtime library functions like

scanf( ) and **atof( )**. There are some cases in which the reference to the **float** is a bit obscure and the compiler does not detect the need for the emulator. The most common is using **scanf( )** to read a **float** in an array of structures as shown in our program. How can we force the formats to be linked? That's where the **linkfloat( )** function comes in. It forces linking of the floating-point emulator into an application. There is no need

12.6 Additional Features of Structures

Let us now explore the intricacies of structures with a view of programming convenience. We would highlight these intricacies with suitable examples:

12.6.1 Assigning Structure Variable

The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator. It is not necessary to copy the structure elements piece-meal. Obviously, programmers prefer assignment to piece-meal copying. This is shown in the following example.

```
void main( )

{

struct employee

{

char name[10] ;

int age ;

float salary ;

} ;

struct employee e1 = { "Sanjay", 30, 5500.50 } ;

struct employee e2, e3 ;

/* piece-meal copying */

strcpy ( e2.name, e1.name ) ;

e2.age = e1.age ;


e2.salary = e1.salary ;

/* copying all elements at one go */

e3 = e2 ;

printf ( "\n%s %d %f", e1.name, e1.age, e1.salary ) ;

printf ( "\n%s %d %f", e2.name, e2.age, e2.salary ) ;

printf ( "\n%s %d %f", e3.name, e3.age, e3.salary ) ;

}
```

The output of the program would be...

```
Sanjay 30 5500.500000

Sanjay 30 5500.500000

Sanjay 30 5500.500000
```

Ability to copy the contents of all structure elements of one variable into the corresponding elements of another structure variable is rather surprising, since C does not allow assigning the contents of one array to another just by equating the two. As we saw earlier, for copying arrays we have to copy the contents of the array element by element.

This copying of all structure elements at one go has been possible only because the structure elements are stored in contiguous memory locations. Had this not been so, we would have been required to copy structure variables element by element. And who knows, had this been so, structures would not have become popular at all.

12.6.2 Nesting of Structure

One structure can be nested within another structure. Using this facility complex data types can be created. The following program shows nested structures at work.

```
main( )

{

struct address

{

char phone[15] ;

char city[25] ;

int pin ;

} ;

struct emp

{

char name[25] ;

struct address a ;

} ;

struct emp e = { "jeru", "531046", "nagpur", 10 };

printf ( "\nname = %s phone = %s", e.name, e.a.phone ) ;

printf ( "\ncity = %s pin = %d", e.a.city, e.a.pin ) ;

}
```

And here is the output...

```
name = jeru phone = 531046
```

city = nagpur pin = 10

Notice the method used to access the element of a structure that is part of another structure. For this the dot operator is used twice, as in the expression,

e.a.pin or e.a.city

Of course, the nesting process need not stop at this level. We can nest a structure within a structure, within another structure, which is in still another structure and so on... till the time we can comprehend the structure ourselves. Such construction however gives rise to variable names that can be surprisingly self descriptive, for example:

maruti.engine.bolt.large.qty

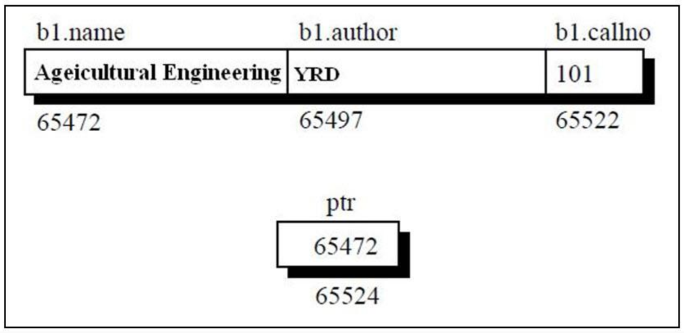
This clearly signifies that we are referring to the quantity of large sized bolts that fit on an engine of a maruti car.

12.6.3 Pointer to Structure

The way we can have a pointer pointing to an **int**, or a pointer pointing to a **char**, similarly we can have a pointer pointing to a **struct**. Such pointers are known as 'structure pointers'. Let us look at a program that demonstrates the usage of a structure pointer.

```
void main( )

{

struct book

{

char name[25] ;

char author[25] ;

int callno ;

} ;

struct book b1 = { "Agricultural Engineering", "YRD", 101 } ;

struct book *ptr ;

ptr = &b1 ;

printf ( "\n%s %s %d", b1.name, b1.author, b1.callno ) ;

printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno ) ;

}
```

The first **printf( )** is as usual. The second **printf( )** however is peculiar. We can't use **ptr.name** or **ptr.callno** because **ptr** is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left. In such cases C provides an operator ->, called an arrow operator to refer to the structure elements. Remember that on the left hand side of the '.' structure operator, there must always be a structure variable, whereas on the left hand side of the '->' operator there must always be a pointer to a structure. The arrangement of the structure variable an



Can we not pass the address of a structure variable to a function? We can. The following program demonstrates this.

```
/* Passing address of a structure variable */

struct book

{

char name[25] ;

char author[25] ;

int callno ;

} ;

main( )

{

struct book b1 = { "Let us C", "YPK", 101 } ;

display ( &b1 ) ;

}

display ( struct book *b )

{

printf ( "\n%s %s %d", b->name, b->author, b->callno ) ;

}
```

And here is the output...

Agricultural Engineering YRD 101

Again note that to access the structure elements using pointer to a structure we have to use the **'->'** operator.

Also, the structure **struct book** should be declared outside **main( )** such that this data type is available to **display( )** while declaring pointer to the structure.

12.6 Union

Union is user defined data type used to stored data under unique variable name at single memory location.

Union is similar to that of stucture. Syntax of union is similar to stucture. But the major **difference between structure and union is 'storage.'** In structures, each member has its own storage location, whereas all the members of union use the same location. Union contains many members of different types, it can handle only one member at a time.

To declare union data type, '**union**' keyword is used.

Union holds value for one data type which requires larger storage among their members.

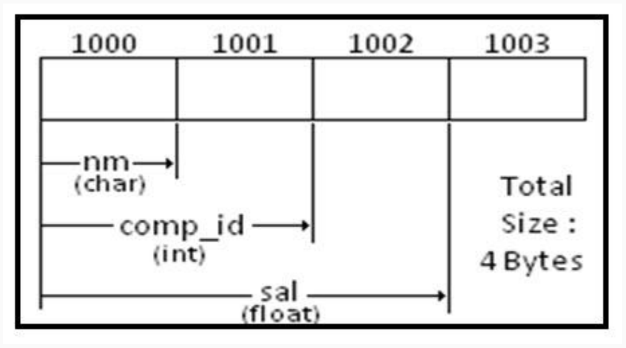
Syntax:

```
    union union_name

    {

        <data-type> element 1;

        <data-type> element 2;

        <data-type> element 3;

    }union_variable;
```

Example:

```
    union techno

    {

        int comp_id;

        char nm;

        float sal;

    }tch;
```

In above example, it declares tch variable of type union. The union contains three members as data type of int, char, float. We can use only one of them at a time.



To access union members, we can use the following syntax.

```
#include <stdio.h>

#include <conio.h>

union techno

{

        int id;

        char nm[50];

}tch;

void main()

{

        clrscr();

        printf("\n\ Enter developer id : ");

        scanf("%d", &tch.id);

        printf("\n Enter developer name : ");

        scanf("%s", tch.nm);

        printf("\n Developer ID : %d", tch.id);//Garbage

        printf("\n\ Developed By : %s", tch.nm);

        getch();
```

Lesson - 13 Pointer

13.1 INTRODUCTION

Which feature of C do beginners find most difficult to understand? The answer is easy: pointers. Other languages have pointers but few use them so frequently as C does. And why not? It is C's clever use of pointers that makes it the excellent language it is.

The difficulty beginners have with pointers has much to do with C's pointer terminology than the actual concept. For instance, when a C programmer says that a certain variable is a "pointer", what does that mean? It is hard to see how a variable can point to something, or in a certain direction.

It is hard to get a grip on pointers just by listening to programmer's jargon. In our discussion of C pointers, therefore, we will try to avoid this difficulty by explaining pointers in terms of programming concepts we already understand. The first thing we want to do is explain the rationale of C's pointer notation.

13.2 Pointer Notation

Consider the declaration,

int i = 3 ;

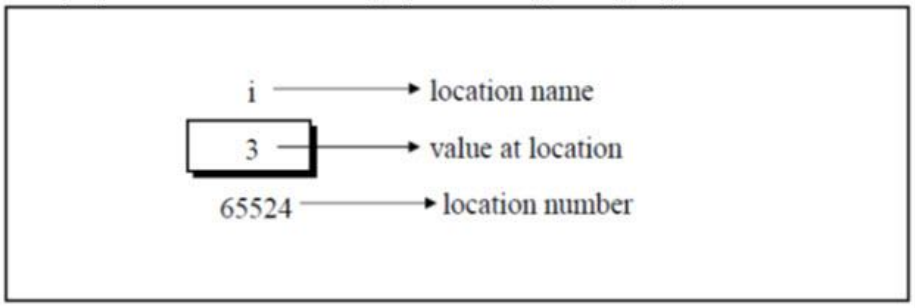
This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name **i** with this memory location.

(c) Store the value 3 at this location.

We may represent **i**'s location in memory by the following memory map.



We see that the computer has selected memory location 65524 as the place to store the value 3. The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is, **i**'s address in memory is a number.

We can print this address number through the following program:

```
void main( )

{

int i = 3 ;

printf ( "\nAddress of i = %u", &i ) ;

printf ( "\nValue of i = %d", i ) ;

}
```

The output of the above program would be:

```
Address of i = 65524

Value of i = 3
```

Look at the first **printf( )** statement carefully. '&' used in this statement is C's 'address of' operator. The expression **&i** returns the address of the variable **i**, which in this case happens to be 65524. Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using **%u**, which is a format specifier for printing an unsigned integer. We have been using the '&' operator all the time in the **scanf( )** statement.

The other pointer operator available in C is '*****', called 'value at address' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

Observe carefully the output of the following program:

```
void main( )

{

int i = 3 ;

printf ( "\nAddress of i = %u", &i ) ;

printf ( "\nValue of i = %d", i ) ;

printf ( "\nValue of i = %d", *( &i ) ) ;
```

The output of the above program would be:

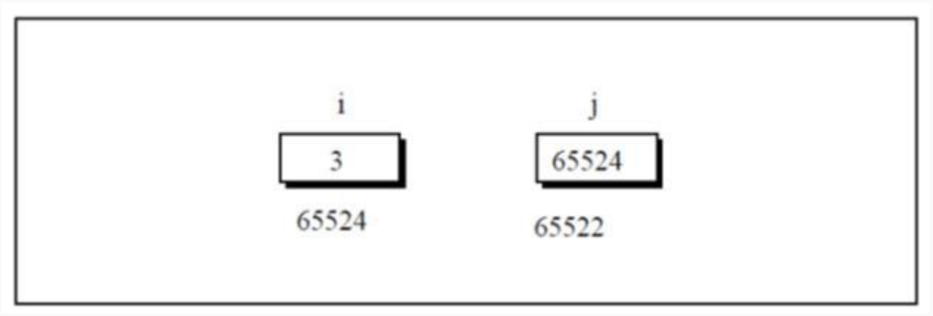
```
Address of i = 65524

Value of i = 3

Value of i = 3
```

Note that printing the value of ***(&i)** is same as printing the value of **i**.

The expression **&i** gives the address of the variable **i**. This address can be collected in a variable, by saying,

j = &i ;

But remember that **j** is not an ordinary variable like any other integer variable. It is a variable that contains the address of other variable (**i** in this case). Since **j** is a variable the compiler must provide it space in the memory. Once again, the following memory map would illustrate the contents of **i** and **j**.



But wait, we can't use **j** in a program without declaring it. And since **j** is a variable that contains the address of **i**, it is declared as,

int *j ;

This declaration tells the compiler that **j** will be used to store the address of an integer value. In other words **j** points to an integer. How do we justify the usage of * in the declaration,

int *j ;

Let us go by the meaning of *. It stands for 'value at address'. Thus, **int *j** would mean, the value at the address contained in **j** is an **int**.

Here is a program that demonstrates the relationships we have been discussing.

```
void main( )

{

int i = 3 ;

int *j ;

j = &i ;

printf ( "\nAddress of i = %u", &i ) ;

printf ( "\nAddress of i = %u", j ) ;
printf ( "\nAddress of j = %u", &j ) ;

printf ( "\nValue of j = %u", j ) ;

printf ( "\nValue of i = %d", i ) ;

printf ( "\nValue of i = %d", *( &i ) ) ;

printf ( "\nValue of i = %d", *j ) ;

}
```

The output of the above program would be:

```
Address of i = 65524

Address of i = 65524

Address of j = 65522

Value of j = 65524

Value of i = 3

Value of i = 3

Value of i = 3
```

Work through the above program carefully, taking help of the memory locations of **i** and **j** shown earlier. This program summarizes everything that we have discussed so far. If you don't understand the program's output, or the meanings of **&i**, **&j**, ***j** and ***(&i)**, re-read the last few pages. Everything we say about C pointers from here onwards will depend on your understanding these expressions thoroughly.

Look at the following declarations,

```
int *alpha ;

char *ch ;

float *s ;
```

Here, **alpha, ch** and **s** are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always contain whole numbers. Now we can put these two facts together and say—pointers are variables that contain addresses, and since addresses are always whole numbers, pointers would always contain whole numbers.

The declaration **float *s** does not mean that **s** is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char**

***ch** means that **ch** is going to contain the address of a char value. Or in other words, the value at address stored in **ch** is going to be a **char**.

The concept of pointers can be further extended. Pointer, we know is a variable that contains address of another variable. Now this variable itself might be another pointer. Thus, we now have a pointer that contains another pointer's address. The following example should make this point clear.

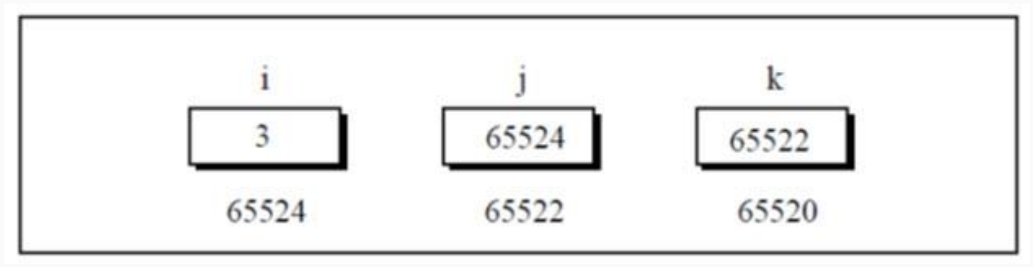
```
void main( )

{

int i = 3, *j, **k ;

j = &i ;

k = &j ;

printf ( "\nAddress of i = %u", &i ) ;

printf ( "\nAddress of i = %u", j ) ;

printf ( "\nAddress of i = %u", *k ) ;

printf ( "\nAddress of j = %u", &j ) ;

printf ( "\nAddress of j = %u", k ) ;

printf ( "\nAddress of k = %u", &k ) ;

printf ( "\nValue of j = %u", j ) ;

printf ( "\nValue of k = %u", k ) ;

printf ( "\nValue of i = %d", i ) ;

printf ( "\nValue of i = %d", * ( &i ) ) ;

printf ( "\nValue of i = %d", *j ) ;

printf ( "\nValue of i = %d", **k ) ;

}
```

The output of the above program would be:

```
Address of i = 65524

Address of i = 65524

Address of i = 65524
Address of j = 65522

Address of j = 65522

Address of k = 65520

Value of j = 65524

Value of k = 65522

Value of i = 3

Value of i = 3

Value of i = 3

Value of i = 3
```

Following Figure would help you in tracing out how the program prints the above output.

Remember that when you run this program the addresses that get printed might turn out to be something different than the ones shown in the figure. However, with these addresses too the relationship between **i**, **j** and **k** can be easily established.



Observe how the variables **j** and **k** have been declared,

```
int i, *j, **k ;
```

Here, **i** is an ordinary **int**, **j** is a pointer to an **int** (often called an integer pointer), whereas **k** is a pointer to an integer pointer. We can extend the above program still further by creating a pointer to a pointer to an integer pointer. In principle, you would agree that likewise there could exist a pointer to a pointer to a pointer to a pointer to a pointer. There is no limit on how far can we go on extending this definition. Possibly, till the point we can comprehend it. And that point of comprehension is usually a pointer to a pointer. Beyond this one rarely requires to extend the definition of a pointer. But just in case...

13.3 Pointer and Function

Having had the first tryst with pointers let us now get back to what we had originally set out to learn—the two types of function calls—call by value and call by reference. Arguments can generally be passed to functions in one of the two ways:

- sending the values of the arguments

- sending the addresses of the arguments

In the first method the 'value' of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. The following program illustrates the 'Call by Value'.

```
main( )

{

int a = 10, b = 20 ;

swapv ( a, b ) ;

printf ( "\na = %d b = %d", a, b ) ;

}

swapv ( int x, int y )

{

int t ;

t = x ;

x = y ;

y = t ;

printf ( "\nx = %d y = %d", x, y ) ;

}
```

The output of the above program would be:

```
x = 20 y = 10

a = 10 b = 20
```

Note that values of **a** and **b** remain unchanged even after exchanging the values of **x** and **y**.

In the second method (call by reference) the addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

```
void main( )

{
```

```
int a = 10, b = 20 ;

swapr ( &a, &b ) ;

printf ( "\na = %d b = %d", a, b ) ;

}

swapr( int *x, int *y )

{

int t ;

t = *x ;

*x = *y ;

*y = t ;

}
```

The output of the above program would be:

```
a = 20 b = 10
```

Note that this program manages to exchange the values of **a** and **b** using their addresses stored in **x** and **y**.

Usually in C programming we make a call by value. This means that in general you cannot alter the actual arguments. But if desired, it can always be achieved through a call by reference.

Using a call by reference intelligently we can make a function return more than one value at a time, which is not possible ordinarily. This is shown in the program given below.

```
void main()

{

int radius ;

float area, perimeter ;

printf ( "\nEnter radius of a circle " ) ;

scanf ( "%d", &radius ) ;

areaperi ( radius, &area, &perimeter ) ;

printf ( "Area = %f", area ) ;
printf ( "\nPerimeter = %f", perimeter ) ;

}

areaperi ( int r, float *a, float *p )

{

*a = 3.14 * r * r ;

*p = 2 * 3.14 * r ;

}
```

And here is the output...

```
Enter radius of a circle 5

Area = 78.500000

Perimeter = 31.400000
```

Here, we are making a mixed call, in the sense, we are passing the value of **radius** but, addresses of **area** and **perimeter**. And since we are passing the addresses, any change that we make in values stored at addresses contained in the variables **a** and **p**, would make the change effective in **main( )**. That is why when the control returns from the function **areaperi( )** we are able to output the values of **area** and **perimeter**.

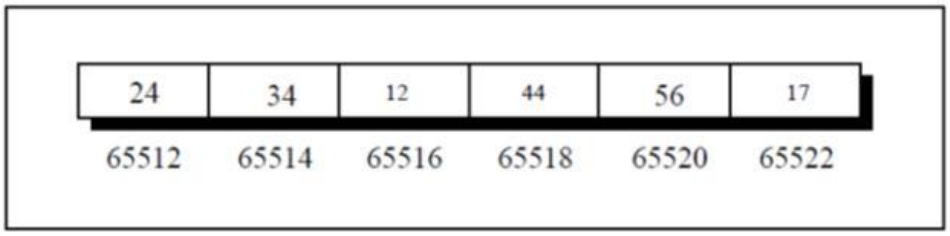
Thus, we have been able to indirectly return two values from a called function, and hence, have overcome the limitation of the **return** statement, which can return only one value from a function at a time.

13.4 Pointer and Array

We will try to correlate the following two facts, which we have learnt before using Pointers and Array:

- Array elements are always stored in contiguous memory locations.
- A pointer when incremented always points to an immediately next location of its type.

Suppose we have an array **num[ ]** = { 24, 34, 12, 44, 56, 17 }. The following figure shows how this array is located in memory.



Here is a program that prints out the memory locations in which the elements of this array are stored.

```
void main( )

{

int num[ ] = { 24, 34, 12, 44, 56, 17 } ;

int i ;

for ( i = 0 ; i <= 5 ; i++ )

{

 printf ( "\nelement no. %d ", i ) ;

printf ( "address = %u", &num[i] ) ;

 }

 }
```

The output of this program would look like this:

```
element no. 0 address = 65512

element no. 1 address = 65514

element no. 2 address = 65516

element no. 3 address = 65518

element no. 4 address = 65520

element no. 5 address = 65522
```

Note that the array elements are stored in contiguous memory locations, each element occupying two bytes, since it is an integer array. When you run this program, you may get different addresses, but what is certain is that each subsequent address would be 2 bytes (4 bytes under Windows/Linux) greater than its immediate predecessor.

Our next two programs show ways in which we can access the elements of this array.

```
void main()

{

int num[ ] = { 24, 34, 12, 44, 56, 17 } ;

 int i ;
for ( i = 0 ; i <= 5 ; i++ )

{

 printf ( "\naddress = %u ", &num[i] ) ;

 printf ( "element = %d", num[i] ) ;

 }

}
```

The output of this program would be:

```
address = 65514 element = 34

address = 65516 element = 12

address = 65518 element = 44

address = 65520 element = 56

address = 65522 element = 17
```

This method of accessing array elements by using subscripted variables is already known to us. This method has in fact been given here for easy comparison with the next method, which accesses the array elements using pointers.

```
Void main( )

{

int num[ ] = { 24, 34, 12, 44, 56, 17 } ;

int i, *j ;

j= &num[0] ; /* assign address of zeroth element */

for ( i = 0 ; i <= 5 ; i++ )

{

printf ( "\naddress = %u ", j ) ;

printf ( "element = %d", *j ) ; j++ ; /* increment pointer to point to next location */

}
```

The output of this program would be:

```
address = 65512 element = 24
address = 65514 element = 34

address = 65516 element = 12

address = 65518 element = 44

address = 65520 element = 56

address = 65522 element = 17
```

In this program, to begin with we have collected the base address of the array (address of the 0th element) in the variable **j** using the statement,

j = &num[0] ; /* assigns address 65512 to j */

When we are inside the loop for the first time, **j** contains the address 65512, and the value at this address is 24. These are printed using the statements,

printf ("\naddress = %u ", j) ; printf ("element = %d", *j) ;

On incrementing **j** it points to the next memory location of its type (that is location no. 65514). But location no. 65514 contains the second element of the array, therefore when the **printf( )** statements are executed for the second time they print out the second element of the array and its address (i.e. 34 and 65514)... and so on till the last element of the array has been printed.

Obviously, a question arises as to which of the above two methods should be used when? Accessing array elements by pointers is **always** faster than accessing them by subscripts. However, from the point of view of convenience in programming we should observe the following:

Array elements should be accessed using pointers if the elements are to be accessed in a fixed order, say from beginning to end, or from end to beginning, or every alternate element or any such definite logic.

Instead, it would be easier to access the elements using a subscript if there is no fixed logic in accessing the elements. However, in this case also, accessing the elements by pointers would work faster than subscripts.

Module-4 Data Structures

Lesson -14 Linked List

14.1 Introduction

United we stand, divided we fall! This has been proved many numbers of times. More united and connected we are, more is the flexibility and scalability. Same is true with linked list. This is perhaps one data structure that has been used at more number of places in computing than you can count. But, beware, they are not simple. But the flexibility and performance they offer is worth the pain of learning them.

For storing similar data in memory we can use either an array or a linked list. Arrays are simple to understand and elements of an array are easily accessible. But arrays suffer from the following limitations:

Arrays have a fixed dimension. Once the size of an array is decided it cannot be increased or decreased during execution. For example, if we construct an array of 100 elements and then try to stuff more than 100 elements in it, our program may crash. On the other hand, if we use only 10 elements then the space for balance 90 elements goes waste.

Array elements are always stored in contiguous memory locations. At times it might so happen that enough contiguous locations might not be available for the array that we are trying. to create. Even though the total space requirement of the array can be met through a combination of non-contiguous blocks of memory, we would still not be allowed to create the array.

Operations like insertion of a new element in an array or deletion of an existing element from the array are pretty tedious. This is because during insertion or deletion each element after the specified position has to be shifted one position to the right (in case of insertion) or one position to the left (in case of deletion).

Linked list overcomes all these disadvantages. A linked list can grow and shrink in size during its lifetime. In other words, there is no maximum size of a linked list. The second advantage of linked lists is that, as nodes (elements) are stored at different memory locations it hardly happens that. we fall short of memory when required. The third advantage is that, unlike arrays, while inserting or deleting the nodes of the linked list, shifting of nodes is not required.

14.2 What Is a Linked List

Linked list is a very common data structure often used to store similar data in memory. While the elements of an array occupy contiguous memory locations, those of a linked list are not constrained to be stored in adjacent locations. The individual elements are stored "somewhere" in memory, rather like a family dispersed, but still bound together. The order of the elements is maintained by explicit links between them. For instance, the integer number can be stored in a linked list as shown in figure 14.1.

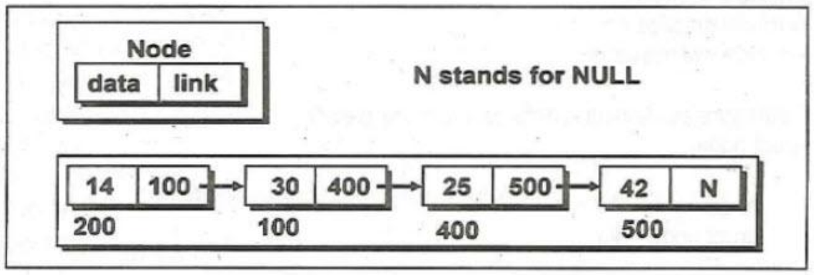


Figure 14.1 – Linked list

Observe that the linked list is a collection of elements called nodes, each of which stores two items of information first is an element of the list and second is a link. A link is a pointer or an address that indicates explicitly the location of the node containing the successor of the list element. In Figure14.1, the arrows represent the links. The **data** part of each node consists of the marks obtained by a student and the **link** part is a pointer to the next node. The **NULL** in the last node indicates that this is the last node in the list.

14.3 Operations on Linked List

There are several operations that we can think of performing on linked list. The following program shows how to build a linked list by adding new nodes at the beginning, at the end or in the middle of the linked list. It also contains a function **display()** which displays all the nodes present in the linked list and a function **del()** which can delete any node in the linked list. Go through the program carefully, a step a time.

```
#include <stdio.h>

#include <conio.h>

#include <alloc.h>

/* structure containing a data part and link part */

struct node

{

        int data ;

        struct node * link ;

} ;

void append ( struct node **, int ) ;

void addatbeg ( struct node **, int ) ;

void addafter ( struct node *, int, int ) ;

void display ( struct node * ) ;
```

```
int count ( struct node * ) ;

void delete ( struct node **, int ) ;

void main( )

{

        struct node *p ;

        p = NULL ;  /* empty linked list */

        printf ( "\nNo. of elements in the Linked List = %d", count ( p ) ) ;

        append ( &p, 14 ) ;

        append ( &p, 30 ) ;

        append ( &p, 25 ) ;

        append ( &p, 42 ) ;

        append ( &p, 17 ) ;

        display ( p ) ;

        addatbeg ( &p, 999 ) ;

        addatbeg ( &p, 888 ) ;

        addatbeg ( &p, 777 ) ;

        display ( p ) ;

        addafter ( p, 7, 0 ) ;

        addafter ( p, 2, 1 ) ;

        addafter ( p, 5, 99 ) ;

        display ( p ) ;

        printf ( "\nNo. of elements in the Linked List = %d", count ( p ) ) ;

        delete ( &p, 99 ) ;

        delete ( &p, 1 ) ;

        delete ( &p, 10 ) ;

        display ( p ) ;
```

```
    printf ( "\nNo. of elements in the Linked List = %d", count ( p ) ) ;
}
/* adds a node at the end of a linked list */
void append ( struct node **q, int num )
{
    struct node *temp, *r ;
    if ( *q == NULL ) /* if the list is empty, create first node */
    {
        temp = malloc ( sizeof ( struct node ) ) ;
        temp -> data = num ;
        temp -> link = NULL ;
        *q = temp ;
    }
    else
    {
        temp = *q ;
        /* go to last node */
        while ( temp -> link != NULL )
            temp = temp -> link ;
        /* add node at the end */
        r = malloc ( sizeof ( struct node ) ) ;
        r -> data = num ;
        r -> link = NULL ;
        temp -> link = r ;
    }
}
```

```
/* adds a new node at the beginning of the linked list */
void addatbeg ( struct node **q, int num )
{
        struct node *temp ;
        /* add new node */
        temp = malloc ( sizeof ( struct node ) ) ;
        temp -> data = num ;
        temp -> link = *q ;
        *q = temp ;
}
/* adds a new node after the specified number of nodes */
void addafter ( struct node *q, int loc, int num )
{
        struct node *temp, *r ;
        int i ;

        temp = q ;
        /* skip to desired portion */
        for ( i = 0 ; i < loc ; i++ )
        {
                temp = temp -> link ;

                /* if end of linked list is encountered */
                if ( temp == NULL )
                {
                        printf ( "\nThere are less than %d elements in list", loc ) ;
                        return ;
                }
        }


        /* insert new node */
        r = malloc ( sizeof ( struct node ) ) ;
        r -> data = num ;
        r -> link = temp -> link ;
        temp -> link = r ;
}
/* displays the contents of the linked list */
void display ( struct node *q )
{
        printf ( "\n" ) ;
        /* traverse the entire linked list */
        while ( q != NULL )
        {
                printf ( "%d ", q -> data ) ;
                q = q -> link ;
        }
}
/* counts the number of nodes present in the linked list */
int count ( struct node * q )
{
        int c = 0 ;
        /* traverse the entire linked list */
```

```
    while ( q != NULL )
    {
        q = q -> link ;
        c++ ;
    }
    return c ;
}
/* deletes the specified node from the linked list */
void delete ( struct node **q, int num )
{
    struct node *old, *temp ;
    temp = *q ;
    while ( temp != NULL )
    {
        if ( temp -> data == num )
        {
            /* if node to be deleted is the first node in the linked list */
            if ( temp == *q )
                *q = temp -> link ;
            /* deletes the intermediate nodes in the linked list */
            else
                old -> link = temp -> link ;
            /* free the memory occupied by the node */
            free ( temp ) ;
            return ;
        }
```

```
                /* traverse the linked list till the last node is reached */
                else
                {
                        old = temp ;  /* old points to the previous node */
                        temp = temp -> link ;  /* go to the next node */
                 }
        }
        printf ( "\nElement %d not found", num ) ;
}
```

Output:

```
14 30 25 42 17
777 888 999 14 30 25 42 17
777 888 999 1 14 30 99 25 42 17 0
No. of elements in the Linked List = 11
Element 10 not found
777 888 999 14 30 25 42 17 0
No. of elements in the Linked List = 9
```

To begin with we have defined a structure for a node. It contains a data part and a link part. The variable **p** has been declared as pointer to a node. We have used this pointer as pointer to the first node in the linked list. No matter how many nodes get added to the linked list, **p** would continue to pointer to the first node in the list. When no node has been added to the list, **p** has been set to **NULL** to indicate that the list is empty.

The append( ) function has to deal with two situations:

The node is being added to an empty list.

The node is being added at the end of an existing list.

In the first case, the condition

```
if ( *q == NULL)
```

gets satisfied. Hence, space is allocated for the node using malloc( ). Data and the link part of.this node are set up using the statements.

temp à data = num ;

temp à link = NULL;

Lastly, p is made to point to this node, since the first node has been added to the list and p must always point to the first node. Note that *q is nothing but equal to p.

In the other case, when the linked list is not empty, the condition

if (*q == NULL)

would fail, since *q (i.e. p is non-NULL). Now temp is made to point to the first node in the list through the statement .

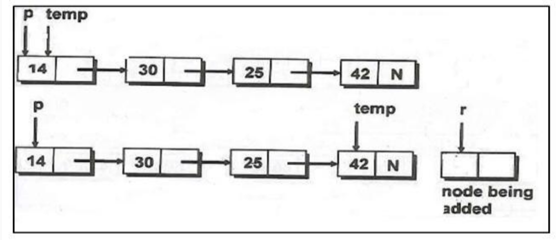
temp = *q;.

Then using temp we have traversed through the entire linked list using the statements

while (temp -> link != NULL)

temp = temp -> link;

The position of the pointers before and after traversing the linked list is shown in following figure.



Each time through the loop the statement temp = temp à link makes temp point to the next node in the list. When temp reaches the last node the condition temp à link != NULL would fail. Once outside the loop we allocate space for the new node through the statement

r =(struet node *) malloc (sizeof (struct node)) ;

Once the space has been allocated for the new node its data part is stuffed with num and the link part with NULL. Note that this node is now going to be the last Node in the list.

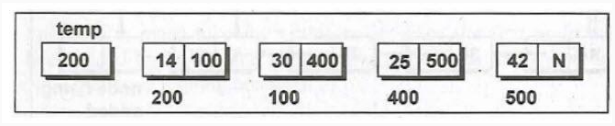
All that now remains to be done is connecting the previous last node with the new last node. The previous last node is being pointed to by temp and the new last node is being pointed to by r.

They are connected through the statement

temp -> link = r ;

this link gets established.

There is often a confusion as to how the statement temp = temp à link makes temp point to the next node in the list. Let us understand this with the help of an example. Suppose in a linked list it contains 4 nodes, temp is pointing at the first node. This is shown in following figure.

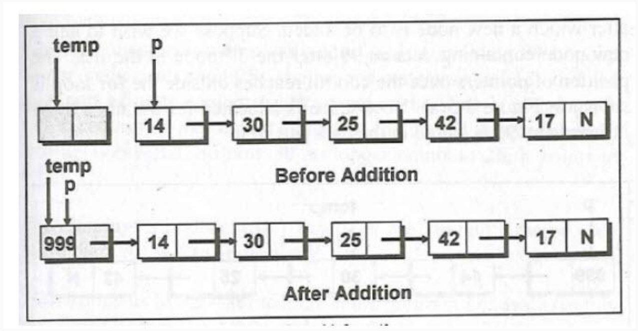


Instead of showing the links to the next node we have shown the addresses of the next node in the link part of each node.

When we execute the statement

temp = temp à link ;

the right hand side yields 100. This address is now stored in temp. As a result, temp starts pointing to the node present at address , 100, In effect the statement has shifted temp so that it has' started pointing to the next node in the list. Let us now understand the addatbeg( ) function. Suppose there are already 5 nodes in the list and we wish to add a new node at the beginning, of this existing linked list. This situation is shown in the following figure.



For adding a new node at the beginning, firstly space is allocated for this node and data is stored in it through the statement

temp **à** data = num ;

Now we need to make the **link** part of this node point to the existing first node. This has been achieved through the statement

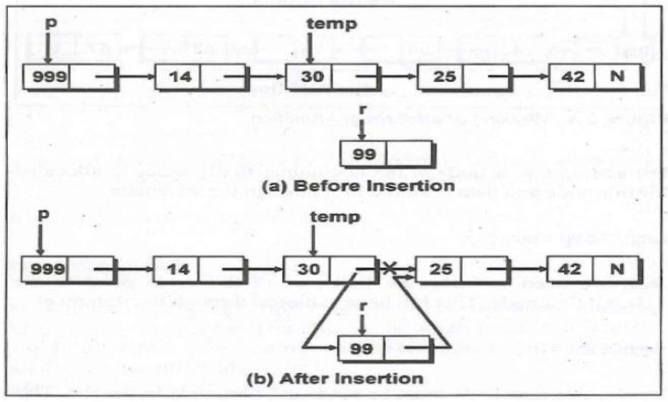
temp **à** link = *q ;

Lastly, this new node must be made the first node in the list. This has been attained through the statement

*q = temp;

The addafter( ) function permits us to add a new node after a specified number of node in the linked list.

To begin with, through a loop we skip the desired number of nodes after which a new node is to be added. Suppose we wish to add a new node containing data as 99 after the 3rd node in the list. The position of pointers once the control reaches outside the for loop is shown in figure (a) given below. Now space is allocated for the node to be inserted and 99 is stored in the data part of it.



All that remains to be done is readjustment of links such that 99 goes in between 30 and 25. This is achieved through the statements

r **à** link = temp **à** link;

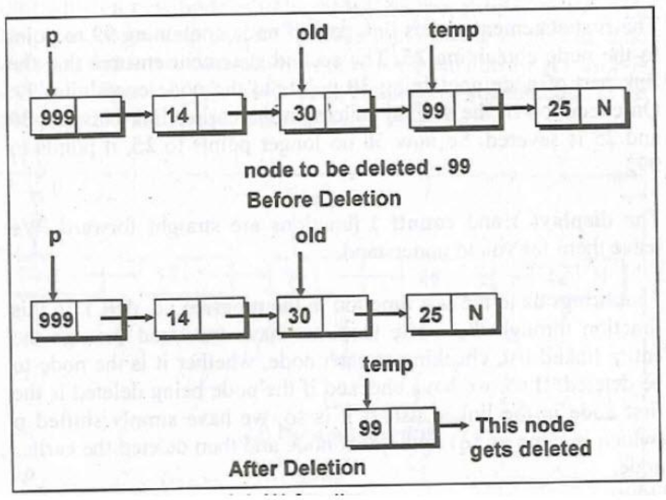
temp **à** link =r ;

The first statement makes link part of node containing **99** to point to the node containing **25**. The second statement ensures that the link part of node .containing **30** points' to the node containing **99**.On execution of the second statement the earlier link between **30** and **25** is severed. So now **30** no longer points to **25**, it points to **99**.

The **display()** and **count()** functions are straight forward. We leave them for you to understand.

That brings us to the last function in the program i.e. **del( )**. In this function through the **while** loop, we have traversed through the entire linked list, checking at each node, whether it is the node to be deleted. If so, we have checked if the node being deleted is the first node in the linked list. If it is so, we have simply shifted **p** (which is same as ***q**) to the next node and then deleted the earlier node.

If the node to be deleted is an intermediate node, then the position of various pointers and links before and after the deletion is shown in following figure.


p
old
temp
999
14
30
99
25
N
node to be deleted - 99
Before Deletion
p
old
999
14
30
25
N
temp
99
This node gets deleted
After Deletion

Lesson- 15 Pointers, Stacks, Push/Pop operations.

15.1 Introduction

Be it items in store, books in a library, or notes in a bank, the moment they become more then handful man starts stacking them neatly. It was natural, that when man started programming data, stack was one of the first structures that he thought of when faced with the problem of maintaining data in an orderly fashion. There is a small difference however, data in a stack is consumed in an orderly fashion; same may not be true in case of wads of nodes. Some of the examples of stack is shown in figure 15.1

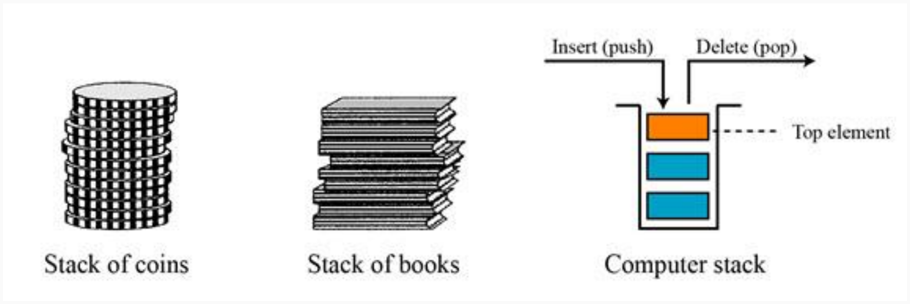


Figure 15.1 Stack Examples

A Stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is often known as **top** of the stack. This situation can be compared to a stack of plates in a cafeteria where every new plate added to the stack is always added at the **top** of the stack. The two changes that can be made to a stack are given special names. When an item is added to a stack, the operation is called **push**, and when an item is removed from the stack, the operation is called **pop**. Stack is also called last-in-first-out (LIFO) list. If the elements are added at continuously to the stack it grows at one end. This is shown in Figure 15.2.

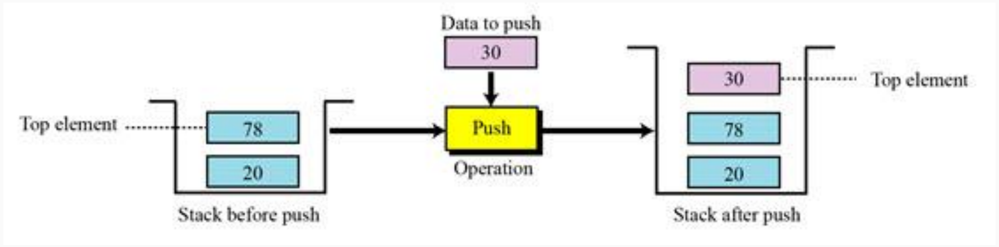


Figure 15.2 Simple Stack with push operation

On deletion of elements the stack shrinks at the same end, as the elements at the top get removed the stack shrinks. This is shown in Figure 15.3.

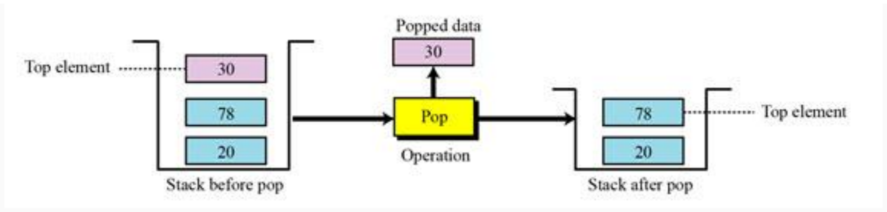


Figure 15.3 Simple Stack with Pop operation

15.2 Operation on Stack

A Stack Generally implements with two basic operations i.e. Push and Pop.

- **Push** – Allows adding an element t the top of the stack.
- **Pop** – Allows removing an element t the top of the stack.

Before making the use of stack in program, it is very important to decide how to represent a stack. There are several ways of representing a stack. Let us see how efficient it is to use an array to represent a stack.

15.3 Stack as an array

Stack contains an ordered collection of elements. An array is used to store ordered list of elements. Hence it would be very easy to manage stack if represent it using an array. However, the problem with an array is that we are required to declare the size of an array before using it in our program. This means the size of an array should be fixed. Stack on other hand does not have any fixed size. It keeps on changing, as the elements in the stack are popped or pushed.

Though an array and a stack are totally different data structures, an array can be used to store the elements of a stack. We can declare an array with a maximum size large enough to manage a stack. As a result the stack can grow or shrink within the space reserved for it. Let us see a program to implements a stack using an array.

```
#include <stdio.h>

#include <conio.h>

#define MAX 10

struct stack

{

        int arr[MAX] ;

        int top ;
```

```
} ;
void initstack ( struct stack * ) ;
void push ( struct stack *, int item ) ;
int pop ( struct stack * ) ;

void main( )
{
        struct stack s ;
        int i ;
        clrscr( ) ;
        initstack ( &s ) ;
        push ( &s, 11 ) ;
        push ( &s, 23 ) ;
        push ( &s, -8 ) ;
        push ( &s, 16 ) ;
        push ( &s, 27 ) ;
        push ( &s, 14 ) ;
        push ( &s, 20 ) ;
        push ( &s, 39 ) ;
        push ( &s, 2 ) ;
        push ( &s, 15 ) ;
        push ( &s, 7 ) ;
        i = pop ( &s ) ;
        printf ( "\n\nItem popped: %d", i ) ;
        i = pop ( &s ) ;
        printf ( "\nItem popped: %d", i ) ;
        i = pop ( &s ) ;
        printf ( "\nItem popped: %d", i ) ;

        i = pop ( &s ) ;
        printf ( "\nItem popped: %d", i ) ;

        i = pop ( &s ) ;
        printf ( "\nItem popped: %d", i ) ;

        getch( ) ;
}
/* intializes the stack */
void initstack ( struct stack *s )
{
        s -> top = -1 ;
}

/* adds an element to the stack */
void push ( struct stack *s, int item )
{
        if ( s -> top == MAX - 1 )
        {
                printf ( "\nStack is full." ) ;
                return ;
        }
        s -> top++ ;
        s -> arr[s ->top] = item ;

}


/* removes an element from the stack */

int pop ( struct stack *s )

{

        int data ;

        if ( s -> top == -1 )

        {

                printf ( "\nStack is empty." ) ;

                return NULL ;

        }

        data = s -> arr[s -> top] ;

        s -> top-- ;

        return data ;

}
```

Output:

```
Stack is full

Item popped: 15

Item popped: 2

Item popped: 39

Item popped: 20

Item popped: 14
```

Here to begin with we have defined structure called **stack**. The **push()** and **pop()** functions are used respectively to add and delete items from the stack. The actual storage of stack elements is done in an array **arr**. The variable **top** is an index into that array. It contains a value where addition or deletion is going to take place in the array, and thereby in the stack. To indicate that the stack is empty to begin with, the variable **top** is set with value -1 calling the function **initstack()**.

Every time an element is added to the stack, it is verified whether such an addition is possible at all. If it is not then the message " Stack is full" is reported. Since w have declared the array to hold 10 elements, the stack would be considered full if the value of t**op** becomes equal to 9.

In **main()** we called **push()** function to add 11 elements to the stack. The value of **top** would become 9 after adding the 10 elements. As a result, the 11th element 7 would not get added to the stack. Lastly, we have removed few elements from the stack by calling the **pop()** function.

15.4 Stack as a Linked list

In the earlier section we had used arrays to store the elements that get added to the stack. However, when implemented as an array, that its size cannot be increased or decreased once it is declared, As a result, one ends up reserving either too much space or too less space for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list. In case of linked stack we shall push and pop nodes from one end of a linked list.

The stack as linked list is represented as a single connected list. Each node in the linked list contains the data nada pointer that gives address of the next node in the list. The node in the list is a structure as shown below:

```
struct node
{
        <data type> data;
        node *link;
};
```

Where <data type> indicates that the data can be any type like **int, float, char** etc, and **link**, is a pointer to the next node in the list. The pointer to beginning of the list serves the purpose of the top of the stack. Figure 15.4 shows the linked list representation of the stack.

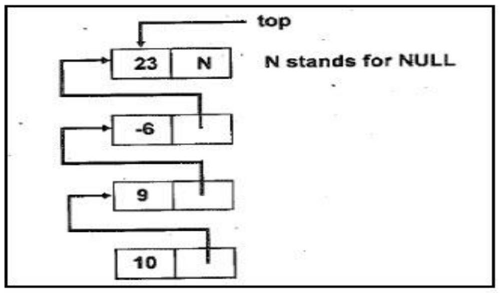


Figure 15.4 Linked list representation of stack.

Let us now see a program that implements stack as a linked list:

```
#include <stdio.h>

#include <conio.h>

#include <alloc.h>

 /* structure containing data part and linkpart */

struct node

{

        int data ;

        struct node *link ;

} ;

 void push ( struct node **, int ) ;

int pop ( struct node ** ) ;

void delstack ( struct node ** ) ;

 void main( )

{

        struct node *s = NULL ;

        int i ;

        clrscr( ) ;

        push ( &s, 14 ) ;

        push ( &s, -3 ) ;

        push ( &s, 18 ) ;

        push ( &s, 29 ) ;

        push ( &s, 31 ) ;

        push ( &s, 16 ) ;

        i = pop ( &s ) ;

        printf ( "\nItem popped: %d", i ) ;
```

```
    i = pop ( &s ) ;
     printf ( "\nItem popped: %d", i ) ;
    i = pop ( &s ) ;
    printf ( "\nItem popped: %d", i ) ;
     delstack ( &s ) ;
     getch( ) ;
}
/* adds a new node to the stack as linked list */
void push ( struct node **top, int item )
{
    struct node *temp ;
    temp = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
     if ( temp == NULL )
        printf ( "\nStack is full." ) ;
     temp -> data = item ;
    temp -> link = *top ;
    *top = temp ;
}
/* pops an element from the stack */
int pop ( struct node **top )
{
    struct node *temp ;
    int item ;
    if ( *top == NULL )
    {
        printf ( "\nStack is empty." ) ;
```

```
                return NULL ;
        }
        temp = *top ;
        item = temp -> data ;
        *top = ( *top ) -> link ;

        free ( temp ) ;
        return item ;
}
/* deallocates memory */
void delstack ( struct node **top )
{
        struct node *temp ;

        if ( *top == NULL )
                return ;
         while ( *top != NULL )
        {
                temp = *top ;
                *top = ( *top ) -> link ;
                free ( temp ) ;
        }
}
```

Output:

```
Item popped: 16
Item popped: 31
```

Item popped: 29

Here we designed a structure called **node**. The variables s is a pointer to the structure **node**. Initially s is set to **NULL** to indicate that the stack is empty. In every call to the function **push()** we are creating a new node dynamically. As long as there is enough space for dynamic memory allocation **temp** would never become **NULL**. If the value if temp happens to be NULL then that would be the stage when the stack would become full.

After, creating a new node, the pointer s should point to the newly created item of the list. Hence, we have assigned the address of this new node to s using the pointer **top**. The stack as a linked list would grow as shown in figure 15.5.

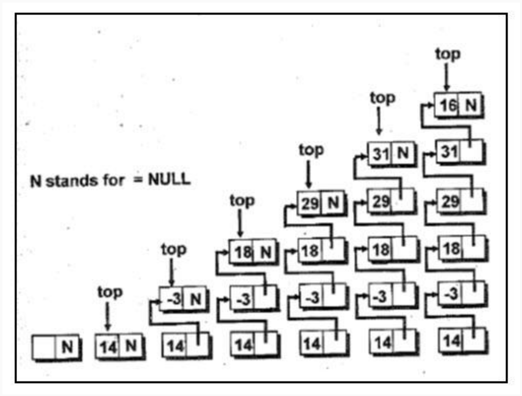


Figure 15.5 – Stack as a linked list after insertion of elements

In the pop() function, first we are checking whether or not a stack is empty. If the stack is empty the the message "Stack is empty" gets displayed. If the stack is nit empty then the topmost item gets removed from the the list. The stack after removing three items from the list would be shown in figure 15.6.

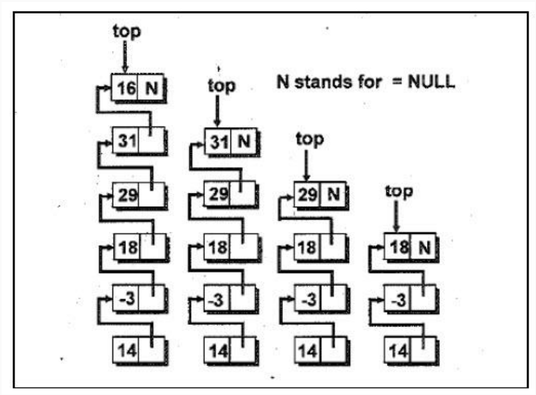


Figure 15.6 – Stack as a linked list after deletion of elements

Lesson -16 Queue

16.1 Introduction

Whether it is a railway reservation counter, a movie theater or print jobs submitted to network printer there is only one way to bring order to chaos from a queue. If you await your turn patiently there is more likelihood that you would get a better service.

Queue is a linear data structure that permits insertion of new element at one end and deletion of element at the other end. The end at which the deletion of an element take place is called **front**, and the end at which insertion of a new element can take place is called **rear**. The deletion or insertion of elements can take place only at the **front** or **rear** end of the list respectively.

The first element that gets added into the queue is the first one to get removed from the list. Hence, queue is also referred to as first-in- first-out (FIFO) list. The name queue comes from the everyday use of the term. Consider a queue of people waiting at a bus stop. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first. The first person in the line is the first person to leave. The figure 16.1 gives a pictorial representation of a queue.

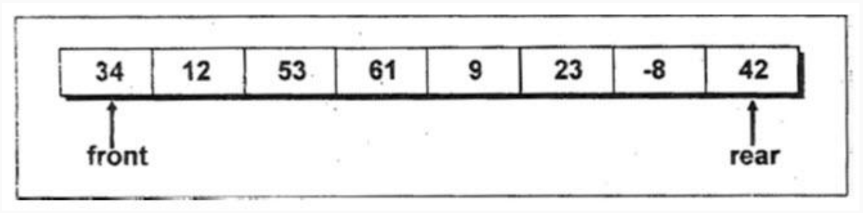


Figure 16.1 – Simple Queue

In Figure 16.1, 34 is the first element and 42 is the last element added to the queue. Similarly, 34 would be the first element to get removed and 42 would be the last element to get removed.

Applications of queue as a data structure are even more common than applications of stacks. While performing tasks on a computer, it is often necessary to wait one's turn before having access to some device or process. Within a computer system there may be queues of tasks waiting for the line printer, or for access to disk storage, or in a time sharing system for use of the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue.

Let us now discuss how a queue can be represented in order to use it in a program.

16.2 Representation of a Queue as an Array

Queue, being a linear data structure can be represented in various ways such as arrays and linked lists. Representing a queue as an array would have the same problem that we discussed in case of stacks. An array is a data structure that can store a fixed number of elements. The size of an array should be fixed before using it. Queue, on the other hand keeps on changing as we remove elements from the front end or add new elements at the rear end. Declaring an array with a maximum size would solve this problem. The maximum size should be large enough for a queue to expand or shrink. Figure 16.2 shows the representation of a queue as an array.

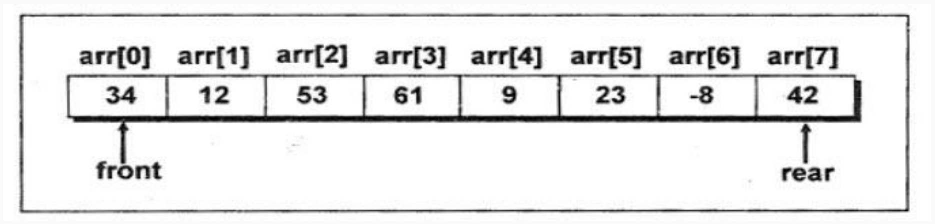


Figure 16.2 – Queue as an array

Let us now see a program that implements queue as an array.

```
#include <stdio.h>

#include <conio.h>

#define MAX 10

void addq ( int *, int, int *, int * ) ;

int delq ( int *, int *, int * ) ;

void main( )

{

        int arr[MAX] ;

        int front = -1, rear = -1, i ;

        clrscr( ) ;

        addq ( arr, 23, &front, &rear ) ;

        addq ( arr, 9, &front, &rear ) ;

        addq ( arr, 11, &front, &rear ) ;

        addq ( arr, -10, &front, &rear ) ;

        addq ( arr, 25, &front, &rear ) ;

        addq ( arr, 16, &front, &rear ) ;

        addq ( arr, 17, &front, &rear ) ;
        addq ( arr, 22, &front, &rear ) ;

        addq ( arr, 19, &front, &rear ) ;

        addq ( arr, 30, &front, &rear ) ;

        addq ( arr, 32, &front, &rear ) ;

        i = delq ( arr, &front, &rear ) ;

        printf ( "\nItem deleted: %d", i ) ;

        i = delq ( arr, &front, &rear ) ;

        printf ( "\nItem deleted: %d", i ) ;

        i = delq ( arr, &front, &rear ) ;

        printf ( "\nItem deleted: %d", i ) ;

        getch( ) ;

}

/* adds an element to the queue */

void addq ( int *arr, int item, int *pfront, int *prear  )

{

        if ( *prear == MAX - 1 )

        {

                printf ( "\nQueue is full." ) ;

                return ;

        }

        ( *prear )++ ;

        arr[*prear] = item ;

        if ( *pfront == -1 )

                *pfront = 0 ;

}

/* removes an element from the queue */
```

```
int delq ( int *arr, int *pfront, int *prear )

{

        int data ;

        if ( *pfront == -1 )

        {

                printf ( "\nQueue is Empty." ) ;

                return NULL ;

        }

        data = arr[*pfront] ;

        arr[*pfront] = 0 ;

        if ( *pfront == *prear )

                *pfront = *prear = -1 ;

        else

                ( *pfront )++ ;


        return  data ;

}
```

Output:

```
Queue is full.

Item deleted: 23

Item deleted: 9

Item deleted: 11
```

Here we have used an array **arr** to maintain the queue. We have also declared two variables **front** and **rear** to monitor the two ends of the queue. The initial values of **front** and **rear** are set to -1, which indicate that the queue is empty. The functions **addq( )** and **delq( )** are used to perform add and delete operations on the queue.

While adding a new element to the queue, first it would be ascertained whether such an addition is possible or not. Since the array indexing begins with 0 the maximum number of elements that can be stored in the queue are **MAX** - 1. If these many elements are already present in the queue then it is reported to be full. If the element can be added to the queue then the value of the variable **rear** is incremented using the pointer **prear** and the new item is stored in the array. The addition of an element to the queue has been illustrated in Figure 16.3.

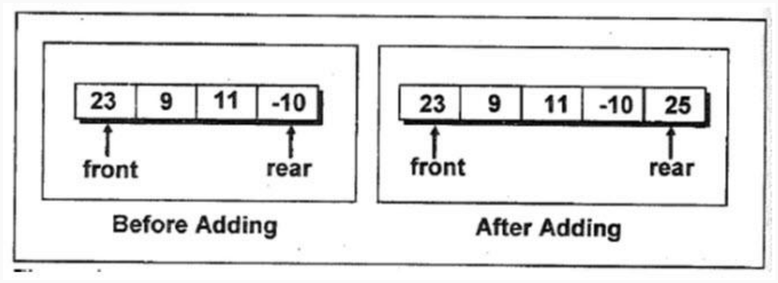


Figure 16.3 - Adding element to a queue

If the item being added to the queue is first element (i.e. if the variable **front** has a value -1) then as soon as the item is added to the queue **front** is set to 0 indicating that the queue is no longer empty.

Let us now see how the **delq( )** function works. Before deleting an element from the queue it is first ascertained whether there are any elements available for deletion. If not, then the queue is reported as empty. Otherwise, an element is deleted form **arr[*pfront]**. The deletion of an element is illustrated in Figure 16.4.

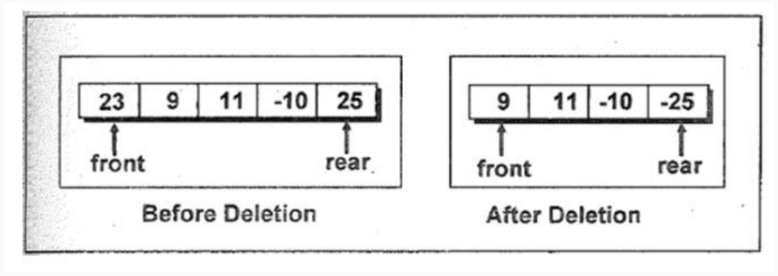


Figure 16.4 - Deleting elements from a queue

Imagine a case where we add 5 elements to the queue. Value of rear would now be 4. Suppose we have not deleted any elements from the queue, then at this stage the value of **front** would be 0. Now suppose we go on deleting elements from the queue. When the fifth element is deleted the queue would fall empty. To make sure that another attempt to delete should be met with an "empty queue" message, in. such a case **front** and **rear** both are reset to -1 to indicate emptiness of the queue.

But this program has got some limitations. Suppose we go on adding elements to the queue till the entire array gets filled. At this stage the value of rear would be **MAX** - 1. Now if we delete 5 elements from the queue, at the end of these deletions the value of **front** would be 5. If now we attempt to add a new element to the queue then it would be reported as full even though in reality the first five slots of the queue are empty. To overcome this situation we can implement a queue as a circular queue, which would discuss later in this chapter.

16.3 Representation of a Queue As a Linked-List

Queue can also be represented using a linked list. As discussed earlier, linked lists do not have any restrictions on the number of elements it can hold. Space for the elements in a linked list is allocated dynamically; hence it can grow as long as there is enough memory available for dynamic allocation. The item in the queue represented as a linked list would be a structure as shown below:

```
struct node

{

<dataType> data;

struct node *link ;

} ;
```

where dataType represents the type of data such as an **int, float, char,** etc. Figure 16.5 shows the representation of a queue as linked list.

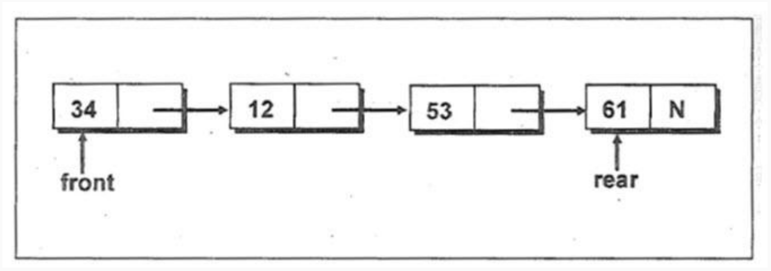


Figure 16.5 – Queue as a linked list

Let us now see a program that implements the queue as a linked list.

```
#include <stdio.h>

#include <conio.h>

struct node

{

        int data ;

        struct node *link ;

} ;

struct queue

{

        struct node *front ;

        struct node *rear ;
```

```
} ;
void initqueue ( struct queue * ) ;
void addq ( struct queue *, int ) ;
int delq ( struct queue * ) ;
void delqueue ( struct queue * ) ;
void main( )
{
        struct queue a ;
        int i ;
        clrscr( ) ;
        initqueue ( &a ) ;
        addq ( &a, 11 ) ;
        addq ( &a, -8 ) ;
        addq ( &a, 23 ) ;
        addq ( &a, 19 ) ;
        addq ( &a, 15 ) ;
        addq ( &a, 16 ) ;
        addq ( &a, 28 ) ;
        i = delq ( &a ) ;
        printf ( "\nItem extracted: %d", i ) ;
        i = delq ( &a ) ;
        printf ( "\nItem extracted: %d", i ) ;
        i = delq ( &a ) ;
        printf ( "\nItem extracted: %d", i ) ;
        delqueue ( &a ) ;
        getch( ) ;
```

```
}
/* initialises data member */
void initqueue ( struct queue *q )
{
        q -> front = q -> rear = NULL ;
}
/* adds an element to the queue */
void addq ( struct queue *q, int item )
{
        struct node *temp ;
        temp = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
        if ( temp == NULL )
                printf ( "\nQueue is full." ) ;
        temp -> data = item ;
        temp -> link = NULL ;
        if ( q -> front == NULL )
        {
                q -> rear = q -> front = temp ;
                return ;
        }
        q -> rear -> link = temp ;
        q -> rear = q -> rear -> link ;
}
/* removes an element from the queue */
int delq ( struct queue * q )
{
        struct node *temp ;
        int item ;
        if ( q -> front == NULL )
        {
                printf ( "\nQueue is empty." ) ;
                return NULL ;
        }
        item = q -> front  -> data ;
        temp = q -> front ;
        q -> front = q -> front -> link ;
        free ( temp ) ;
        return item ;
}
/* deallocates memory */
void delqueue ( struct queue *q )
{
        struct node *temp ;
        if ( q -> front == NULL )
                return ;
        while ( q -> front != NULL )
        {
                temp = q -> front ;
                q -> front = q -> front -> link ;
                free ( temp ) ;
        }
}
```

Output:

```
Item extracted: 11

Item extracted: -8

Item extracted: 23
```

In this program the structure **queue** contains two data member **front** and **rear**, both are of the type pointers to the structure **node** To begin with, the queue is empty hence both **front** and **rear** are set to **NULL**.

The **addq ()** function adds a new element at the rear end of the list. If the element added is the first element, then both **front** and **rear** are made to point to the new node. However, if the element added is not the first element then only **rear** is made to point to the new node, whereas **front** continues to point to the first node in the list.

The **delq()** function removes an element from the list which is at the front end of the list. Removal of an element from the list actually deletes the node to which **front** is pointing. After deletion of a node, **front** is made to point to the next node that comes in the list, whereas **rear** continues to point to the last node in the list.

The function **delqueue()**is called before the function **main()** comes to an end. This is done because the memory allocated for the existing nodes in the list must be de-allocated.

16.4 Circular Queue

The queue that we implemented using an array suffers from one limitation. In that implementation there is a possibility that the queue is reported as full (since **rear** has reached the end of the array), even though in actuality there might be empty slots at the beginning of the queue. To overcome this limitation we can implement the queue as a circular queue. Here as we go on adding elements to the queue and reach the end of the array, the next element is stored in the first slot of the array (provided it is free). More clearly, suppose an array **arr** of **n** elements is used to implement a circular queue. Now if we go on adding elements to the queue we may reach **arr[n-1].** We cannot add any more elements to the queue since we have reached the end of the array. Instead of reporting the queue as full, if some elements in the queue have been deleted then there might be empty slots at the beginning of the queue. In such a case these slots would be filled by new elements being added to the queue. In short just because we have reached the end of the array the queue would not be reported as full. The queue would be reported as full only when all the slots in the array stand occupied. Figure 16.6 shows the pictorial representation of a circular queue.

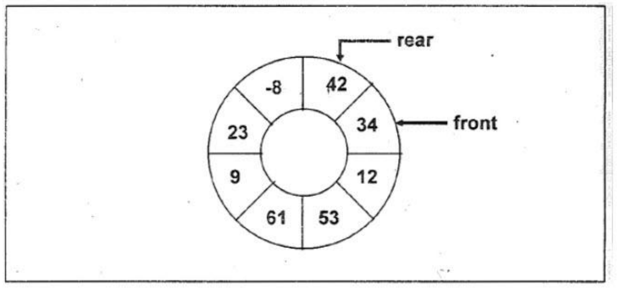


Figure 16.6 – Circular Queue

Let us now see a program that performs the addition and deletion operation on a circular queue.

```
#include <stdio.h>

#include <conio.h>

#define MAX 10

void addq ( int *, int, int *, int * ) ;

int delq ( int *, int *, int * ) ;

void display ( int * ) ;

void main( )

{

        int arr[MAX] ;

        int i, front, rear ;

        clrscr( ) ;

        /* initialise data member */

        front = rear = -1 ;

        for ( i = 0 ; i < MAX ; i++ )

                arr[i] = 0 ;

        addq ( arr, 14, &front, &rear ) ;

        addq ( arr, 22, &front, &rear ) ;

        addq ( arr, 13, &front, &rear ) ;

        addq ( arr, -6, &front, &rear ) ;

        addq ( arr, 25, &front, &rear ) ;

        printf ( "\nElements in the circular queue: " ) ;

        display ( arr ) ;

        i = delq ( arr, &front, &rear ) ;

        printf ( "Item deleted: %d", i ) ;
```

```
    i = delq ( arr, &front, &rear ) ;
    printf ( "\nItem deleted: %d", i ) ;
    printf ( "\nElements in the circular queue after deletion: " ) ;
    display ( arr ) ;
    addq ( arr, 21, &front, &rear ) ;
    addq ( arr, 17, &front, &rear ) ;
    addq ( arr, 18, &front, &rear ) ;
    addq ( arr, 9, &front, &rear ) ;
    addq ( arr, 20, &front, &rear ) ;
    printf ( "Elements in the circular queue after addition: " ) ;
    display ( arr ) ;
    addq ( arr, 32, &front, &rear ) ;
    printf ( "Elements in the circular queue after addition: " ) ;
    display ( arr ) ;
    getch( ) ;
}
/* adds an element to the queue */
void addq ( int *arr, int item, int *pfront, int *prear )
{
    if ( ( *prear == MAX - 1 && *pfront == 0 ) || (  *prear + 1 == *pfront ) )
    {
        printf ( "\nQueue is full." ) ;
        return ;
    }
    if ( *prear == MAX - 1 )
        *prear = 0 ;
        else
                ( *prear )++ ;
        arr[*prear] = item ;
        if ( *pfront == -1 )
                *pfront = 0 ;
}
/* removes an element from the queue */
int delq ( int *arr, int *pfront, int *prear )
{
        int data ;
        if ( *pfront == -1 )
        {
                printf ( "\nQueue is empty." ) ;
                return NULL ;
        }
        data = arr[*pfront] ;
        arr[*pfront] = 0 ;
        if ( *pfront == *prear )
        {
                *pfront = -1 ;
                *prear = -1 ;
        }
        else
        {
                if ( *pfront == MAX - 1 )
                        *pfront = 0 ;
                else
                        ( *pfront )++ ;
        }
        return data ;
}
/* displays element in a queue */
void display ( int * arr )
{
        int i ;
        printf ( "\n" ) ;
        for ( i = 0 ; i < MAX ; i++ )
                printf ( "%d\t", arr[i] ) ;
        printf ( "\n" ) ;
}
```

Output:

```
Elements in the circular queue:
14      22      13      -6      0       0       0       0       0       0
Item deleted: 14
Item deleted: 22
Elements in the circular queue after deletion:
0       0       13      -6      25      0       0       0       0       0
Elements in the circular queue after addition:
0       0       13      -6      25      21      17      18      9       20
Elements in the circular queue after addition:
32      0       13      -6      25      21      17      18      9       20
```

Here the array **arr** is used to store the elements of the circular queue. The functions **addq()** and **delq()** are used to add and remove the elements from the queue respectively. The function **display()** displays the existing elements of the queue. The initial values of **front** and **rear** are set to -1, which indicates that queue is empty.

In **main()** first we have called the **addq()** function 5 times to insert elements in the circular queue. In this function, following cases are considered before adding an element to the queue.

First we have checked whether or not the array is full. The message 'Queue is full' gets displayed if **front** and **rear** are in adjacent locations with **rear** following the **front**.

If the value of **front** is -1 then it indicates that the queue is empty and the element to be added would be the first element in the queue. The values of **front** and **rear** in such a case are set to 0 and the new element gets placed at the 0th position.

It may also happen that some of the positions at the front end of the array are vacant. This happens if we have deleted some elements from the queue, when the value of **rear** is **MAX** -1 and the value of **front** is greater than 0. In such a case the value of **rear** is set to 0 and the element to be added is added at this position.

The element is added at the rear position in case the value of **front** is either equal to or greater than 0 and the value of **rear** is less than **MAX** - I.

Thus, after adding 5 elements the value of **front** and **rear** become 0 and 4 respectively. The **display()** function displays the elements in the queue. Figure 16.7 shows the pictorial representation of a circular queue after adding 5 elements .

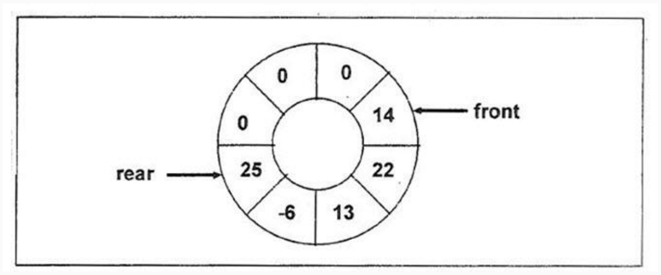


Figure 16.7 – Circular queue after adding elements

Next we have called **delq()** function twice to remove 2 elements from the queue. The following conditions are checked while deleting an element

First we have checked whether or not the queue is empty. The value of **front** in our case is 4, hence an element at the **front** position would get removed.

Next, we have checked if the value of **front** has become equal to **rear**. If it has, then the element we wish to remove is the only element of the queue. On removal of this element the queue would become empty and hence the values of **front** and **rear** are set to -1.

On deleting an element from the queue the value of **front** is set to 0 if it is equal to **MAX** - 1. Otherwise **front** is simply increment by 1. Figure 16.8 shows the pictorial representation of a circular queue after deleting two elements from the queue.

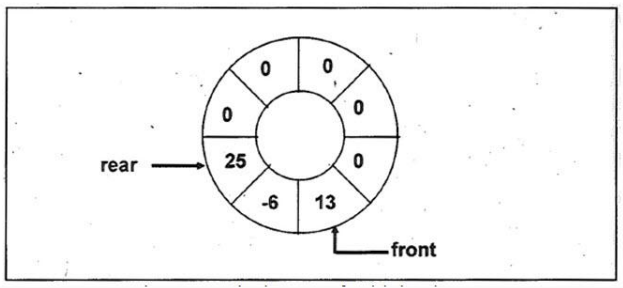


Figure 16.8– Circular queue after deleting elements

Further calls to **addq()** function adds elements 21,17, 18,9, and 20 to the queue. The value of **front** and **rear** at this stage becomes 2 and **MAX** - 1 respectively. One more call **to addq()** function changes the value of rear from **MAX** - 1 to 0 and the element added at the 0th position.