UNIT -1

PROGRAMMING IN C

Programming

To solve a computing problem, its solution must be specified in terms of sequence of computational steps such that they are effectively solved by a human agent or by a digital computer.

Programming Language

1) The specification of the sequence of computational steps in a particular programming language is termed as a program
2) The task of developing programs is called programming
3) The person engaged in programming activity is called programmer

Techniques of Problem Solving

Problem solving an art in that it requires enormous intuitive power & a science for it takes a pragmatic approach.
Here a rough outline of a general problem solving approach.

1) Write out the problem statement include information on what you are to solve & consider why you need to solve the problem
2) Make sure you are solving the real problem as opposed to the perceived problem. To check to see that you define & solve the real problem
3) Draw & label a sketch. Define & name all variables and /or symbols. Show numerical values of variables, if known.
4) Identify & Name
    a. relevant principles, theories & equations
    b. system & subsystems
    c. dependent & independent variables
    d. known & unknowns
    e. inputs & outputs
    f. necessary information
5) List assumptions and approximations involved in solving the problem. Question the assumptions and then state which ones are the most reasonable for your purposes.
6) Check to see if the problem is either under-specified, figure out how to find the missing information. If over-specified, identify the extra information that is not needed.
7) Relate problem to similar problem or experience
8) Use an algorithm

9) Evaluate and examine and evaluate the answer to see it makes sense.

Introduction to C Programming

C is a general-purpose computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system. C is a structured programming language, which means that it allows you to develop programs using well-defined *control structures* (you will learn about *control structures* in the articles to come), 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.

Where is C useful?

C's ability to communicate directly with hardware makes it a powerful choice for system programmers. In fact, popular operating systems such as Unix and Linux are written entirely in C. Additionally, even compilers and interpreters for other languages such as FORTRAN, Pascal, and BASIC are written in C. However, C's scope is not just limited to developing system programs. It is also used to develop any kind of application, including complex business ones. The following is a partial list of areas where C language is used:

- Ø Embedded Systems
- Ø Systems Programming
- Ø Artificial Intelligence
- Ø Industrial Automation
- Ø Computer Graphics
- Ø Space Research

Why you should learn C?

You should learn C because:

- C is simple.
- There are only 32 *keywords* so C is very easy to master. Keywords are words that have special meaning in C language.

- C programs run faster than programs written in most other languages.
- C enables easy communication with computer hardware making it easy to write system programs such as *compilers* and *interpreters*.

WHY WE NEED DATA AND A PROGRAM

Any computer program has two entities to consider, the data, and the program. They are highly dependent on one another and careful planning of both will lead to a well planned and well written program. Unfortunately, it is not possible to study either completely without a good working knowledge of the other. For that reason, this tutorial will jump back and forth between teaching methods of program writing and methods of data definition. Simply follow along and you will have a good understanding of both. Keep in mind that, even though it seems expedient to sometimes jump right into coding the program, time spent planning the data structures will be well spent and the quality of the final program will reflect the original planning

How to run a simple c program

1. Copy Turbo c/c++ in computer

2. Open c:\tc\bin\tc.exe

3. A window appears

4. Select File->new to open a new file

5. Type the following program on editor

```
#include <stdio.h>

void main()

{

        printf(“hello”);

}
```

6. compile the program by pressing ALT+F9

7. Run the program by pressing CTRL +F9

Note:

1. C is case sensitive

2. Always terminate statements with semicolon.

3. A program starts with main()

Explanation of program

#include is known as compiler directive. A compiler directive is a command to compiler to translate the program in a certain way. These statement are not converted into machine language but only perform some other task.

main() is a function which the starting point for complier to start compilation. So a function must contain a main() function.

DETECTION AND CORRECTION OF ERRORS

Syntactic errors and execution errors usually result in the generation of error messages when compiling or executing a program. Error of this type is usually quite easy to find and correct. There are some logical errors that can be very difficult to detect. Since the output resulting from a logically incorrect program may appear to be error free. Logical errors are often hard to find, so in order to find and correct errors of this type is known as logical debugging. To detect errors test a new program with data that will give a known answer. If the correct results are not obtained then the program obviously contains errors even if the correct results are obtained.

Computer Applications: However you cannot be sure that the program is error free, since some errors cause incorrect result only under certain circumstances. Therefore a new program should receive thorough testing before it is considered to be debugged. Once it has been established that a program contains a logical error, some ingenuity may be required to find the error. Error detection should always begin with a thorough review of each logical group of statements within the program. If the error cannot be found, it sometimes helps to set the program aside for a while. If an error cannot be located simply by inspection, the program should be modified to print out certain

intermediate results and then be rerun. This technique is referred to as tracing. The source of error will often become evident once these intermediate calculations have been carefully examined. The greater the amount of intermediate output, the more likely the chances of pointing the source of errors. Sometimes an error simply cannot be located. Some C compilers include a debugger, which is a special program that facilitates the detection of errors in C programs. In particular a debugger allows the execution of a source program to be suspended at designated places, called break points, revealing the values assigned to the program variables and array elements at the time execution stops. Some debuggers also allow a program to execute continuously until some specified error condition has occurred. By examining the values assigned to the variables at the break points, it is easier to determine when and where an error originates.

Linear Programming

Linear program is a method for straightforward programming in a sequential manner. This type of programming does not involve any decision making. General model of these linear programs is:

1. Read a data value
2. Computer an intermediate result
3. Use the intermediate result to computer the desired answer
4. Print the answer
5. Stop

Structured Programming

Structured programming (sometimes known as *modular programming*) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as Ada, Pascal, and dBASE are designed with features that encourage or enforce a logical program structure.

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or sub module, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

Advantages of Structured Programming

1. **Easy to write:**
   Modular design increases the programmer's productivity by allowing them to look at the big picture first and focus on details later.Several Programmers can work on a single, large program, each working on a different module. Studies show structured programs take less time to write than standard programs. Procedures written for one program can be reused in other programs requiring the same task. A procedure that can be used in many programs is said to be reusable.
2. **Easy to debug:**
   Since each procedure is specialized to perform just one task, a procedure can be checked individually. Older unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such programs is cluttered with details and therefore difficult to follow.
3. **Easy to Understand:**
   The relationship between the procedures shows the modular design of the program. Meaningful procedure names and clear documentation identify the task performed by each module. Meaningful variable names help the programmer identify the purpose of each variable.
4. **Easy to Change:**
   Since a correctly written structured program is self-documenting, it can be easily understood by another programmer.

Structured Programming Constructs

It uses only three constructs -

- Sequence (statements, blocks)
- Selection (if, switch)
- Iteration (loops like while and for)

Sequence

- Any valid expression terminated by a semicolon is a statement.
- Statements may be grouped together by surrounding them with a pair of curly braces.
- Such a group is syntactically equivalent to one statement and can be inserted where ever
- One statement is legal.

Selection

The selection constructs allow us to follow different paths in different situations. We may also think of them as enabling us to express decisions.

The main selection construct is:

if (*expression*)
*statement*1
else
*statement*2

*statement*1 is executed if and only if *expression* evaluates to some non-zero number. If *expression* evaluates to 0, *statement*1 is not executed. In that case, *statement*2 is executed.
If and else are independent constructs, in that if can occur without else (but not the reverse).Any else is paired with the most recent else-less if, unless curly braces enforce a different scheme. Note that only curly braces, not parentheses, must be used to enforce the pairing. Parentheses

Iteration

Looping is a way by which we can execute any some set of statements more than one times continuously .In C there are mainly three types of loops are used :

- while Loop
- do while Loop
- For Loop

The control structures are easy to use because of the following reasons:

1) They are easy to recognize
2) They are simple to deal with as they have just one entry and one exit point
3) They are free of the complications of any particular programming language

Modular Design of Programs

One of the key concepts in the application of programming is the design of a program as a set of units referred to as blocks or modules. A style that breaks large computer programs into smaller elements called modules. Each module performs a single task; often a task that needs to be performed multiple times during the running of a program. Each module also stands alone with defined input and output. Since modules are able to be reused they can be designed to be used for multiple programs. By debugging each module and only including it when it performs its defined task, larger programs are easier to debug because large sections of the code have already been evaluated for errors. That usually means errors will be in the logic that calls the various modules.

Languages like Modula-2 were designed for use with modular programming. Modular programming has generally evolved into object-oriented programming.

Programs can be logically separated into the following functional modules:

1) Initialization
2) Input
3) Input Data Validation
4) Processing
5) Output
6) Error Handling
7) Closing procedure

Basic attributes of modular programming:

1) Input
2) Output
3) Function
4) Mechanism
5) Internal data

Control Relationship between modules:

The structure charts show the interrelationships of modules by arranging them at different levels and connecting modules in those levels by arrows. An arrow between two modules means the program control is passed from one module to the other at execution time. The first module is said to call or invoke the lower level modules.There are three rules for controlling the relationship between modules.

1) There is only one module at the top of the structure. This is called the root or boss module.
2) The root passes control down the structure chart to the lower level modules. However, control is always returned to the invoking module and a finished module should always terminate at the root.
3) There can be more than one control relationship between two modules on the structure chart, thus, if module A invokes module B, then B cannot invoke module A.

Communication between modules:

1) **Data**: Shown by an arrow with empty circle at its tail.
2) **Control** : Shown by a filled-in circle at the end of the tail of arrow

Module Design Requirements

A hierarchical or module structure should prevent many advantages in management, developing, testing and maintenance. However, such advantages will occur only if modules fulfill the following requirements.

a) **Coupling:** In computer science, coupling is considered to be the degree to which each program module relies on other modules, and is also the term used to describe connecting two or more systems. Coupling is broken down into loose coupling, tight coupling, and decoupled. Coupling is also used to describe software as well as systems. Also called dependency

Types of Programming Language

Low Level Language

First-generation language is the lowest level computer language. Information is conveyed to the computer by the programmer as binary instructions. Binary instructions are the equivalent of the on/off signals used by computers to carry out operations. The language consists of zeros and ones. In the 1940s and 1950s, computers were programmed by scientists sitting before control panels equipped with toggle switches so that they could input instructions as strings of zeros and ones.

Advantages

- Fast and efficient
- Machine oriented
- No translation required

Disadvantages

- Not portable
- Not programmer friendly

Assembly Language

Assembly or assembler language was the second generation of computer language. By the late 1950s, this language had become popular. Assembly language consists of

letters of the alphabet. This makes programming much easier than trying to program a series of zeros and ones. As an added programming assist, assembly language makes use of mnemonics, or memory aids, which are easier for the human programmer to recall than are numerical codes.

Assembler

An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term *assembly language In other words* An **assembler** is a computer program for translating assembly language — essentially, a mnemonic representation of machine language — into object code. A **cross assembler** (see cross compiler) produces code for one processor, but runs on another.

As well as translating assembly instruction mnemonics into opcodes, assemblers provide the ability to use symbolic names for memory locations (saving tedious calculations and manually updating addresses when a program is slightly modified), and macro facilities for performing textual substitution — typically used to encode common short sequences of instructions to run inline instead of in a subroutine.

High Level Language

The introduction of the compiler in 1952 spurred the development of third-generation computer languages. These languages enable a programmer to create program files using commands that are similar to spoken English. Third-level computer languages have become the major means of communication between the digital computer and its user. By 1957, the International Business Machine Corporation (IBM) had created a language called FORTRAN (FORmula TRANslater). This language was designed for scientific work involving complicated mathematical formulas. It became the first high-level programming language (or "source code") to be used by many computer users.

Within the next few years, refinements gave rise to ALGOL (ALGOrithmic Language) and COBOL (COmmon Business Oriented Language). COBOL is noteworthy because it improved the record keeping and data management ability of businesses, which stimulated business expansion.

Advantages

- Portable or *machine independent*
- Programmer-friendly

Disadvantages

- Not as efficient as low-level languages
- Need to be translated

Examples : C, C++, Java, FORTRAN, Visual Basic, and Delphi.

Interpreter

An **interpreter** is a computer program that executes other programs. This is in contrast to a compiler which does not execute its input program (the source code) but translates it into executable machine code (also called object code) which is output to a file for later execution. It may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced.

It takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

Interpreting code is slower than running the compiled code because the interpreter must analyses each statement in the program each time it is executed and then perform the desired action whereas the compiled code just performs the action. This run-time analysis is known as "interpretive overhead". Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.

COMPILER

A program that translates *source code* into *object code*. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. Thus, a compiler differs from an *interpreter*, which analyzes and executes each line of source code in succession, without looking at the entire program. The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

Every high-level programming language (except strictly interpretive languages) comes with a compiler. In effect, the compiler is the language, because it defines which instructions are acceptable.

Fourth Generation Language

Fourth-generation languages attempt to make communicating with computers as much like the processes of thinking and talking to other people as possible. The problem is that the computer still only understands zeros and ones, so a compiler and interpreter must still convert the source code into the machine code that the computer can understand. Fourth-generation languages typically consist of English-like words and phrases. When they are implemented on microcomputers, some of these languages include graphic devices such as icons and onscreen push buttons for use during programming and when running the resulting application.

Many fourth-generation languages use Structured Query Language (SQL) as the basis for operations. SQL was developed at IBM to develop information stored in relational databases. Examples of fourth-generation languages include PROLOG, an **Artificial Intelligence** language

UNIT-2

FEATURES OF 'C'

C-Language keywords

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

Data Types

A C language programmer has to tell the system before-hand, the type of numbers or characters he is using in his program. These are data types. There are many data types in C language. A C programmer has to use appropriate data type as per his requirement. C language data types can be broadly classified as

- Primary data type
- Derived data type
- User defined data type

Primary data type

All C Compilers accept the following fundamental data types

1.	Integer	int
2.	Character	char
3.	Floating Point	float
4.	Double precision floating point	double
5.	Void	void

The size and range of each data type is given in the table below

DATA TYPE	RANGE OF VALUES
char	-128 to 127
Int	-32768 to +32767
float	3.4 e-38 to 3.4 e+38
double	1.7 e-308 to 1.7 e+308

Integer Type:

Integers are whole numbers with a machine dependent range of values. A good programming language as to support the programmer by giving a control on a range of numbers and storage space. C has 3 classes of integer storage namely short int, int and long int. All of these data types have signed and unsigned forms. A short int requires half the space than normal integer values. Unsigned numbers are always positive and consume all the bits for the magnitude of the number. The long and unsigned integers are used to declare a longer range of values.

Floating Point Types:

Floating point number represents a real number with 6 digits precision. Floating point numbers are denoted by the keyword float. When the accuracy of the floating point number is insufficient, we can use the double to define the number. The double is same as float but with longer precision. To extend the precision further we can use long double which consumes 80 bits of memory space.

Void Type:

Using void data type, we can specify the type of a function. It is a good practice to avoid functions that does not return any values to the calling function.

Character Type:

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

Size and Range of Data Types on 16 bit machine.

TYPE	SIZE (Bits)	Range
Char or Signed Char	8	-128 to 127
Unsigned Char	8	0 to 255
Int or Signed int	16	-32768 to 32767
Unsigned int	16	0 to 65535
Short int or Signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2147483648 to 2147483647
Unsigned long int	32	0 to 4294967295
Float	32	3.4 e-38 to 3.4 e+38
Double	64	1.7e-308 to 1.7e+308
Long Double	80	3.4 e-4932 to 3.4 e+4932

Variable:

Variable is a name of memory location where we can store any data. It can store only single data (Latest data) at a time. In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code, but most are found at the start of each function.

A declaration begins with the type, followed by the name of one or more variables. For example,

DataType Name_of_Variable_Name;

int a,b,c;

Variable Names

Every variable has a name and a value. The name identifies the variable, the value stores data. There is a limitation on what these names can be. Every variable name in C must start with a letter; the rest of the name can consist of letters, numbers and underscore characters. C recognizes upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names.

Examples of legal variable names include:

x	result	outfile	bestyet
x1	x2	out_file	best_yet
power	impetus	gamma	hi_score

It is conventional to avoid the use of capital letters in variable names. These are used for names of constants. Some old implementations of C only use the first 8 characters of a variable name.

Local Variables

Local variables are declared within the body of a function, and can only be used within that function only.

Syntax:

```
void main( ){

int a,b,c;

}

void fun1()

{

int x,y,z;

}
```

Here a,b,c are the local variable of void main() function and it can't be used within fun1() Function. And x, y and z are local variable of fun1().

Global Variable

A global variable declaration looks normal, but is located outside any of the program's functions. This is usually done at the beginning of the program file, but after preprocessor directives. The variable is not declared again in the body of the functions which access it.

Syntax:

```
#include<stdio.h>

int a,b,c;
void main()
{

}
Void fun1()
{
}
```

Here a,b,c are global variable .and these variable cab be accessed (used) within a whole program.

Constants

C constant is usually just the written version of a number. For example 1, 0, 5.73, 12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero - 015.
- Hexadecimal constants are written with a leading 0x - 0x1ae.
- Long constants are written with a trailing L - 890L.

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence.

'\n'	newline
'\t'	tab
'\\ '	backslash
'\''	single quote
'\0'	null (used automatically to terminate character strings).

In addition, a required bit pattern can be specified using its octal equivalent.

'\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes e.g. "Brian and Dennis". The string is actually

stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.

Constant is a special types of variable which can not be changed at the time of execution. Syntax:

```
const int a=20;
```

C Language Operator Precedence Chart

Operator precedence describes the order in which C reads expressions. For example, the expression `a=4+b*2` contains two operations, an addition and a multiplication. Does the C compiler evaluate `4+b` first, then multiply the result by `2`, or does it evaluate `b*2` first, then add `4` to the result? The operator precedence chart contains the answers. Operators higher in the chart have a higher precedence, meaning that the C compiler evaluates them first. Operators on the same line in the chart have the same precedence, and the "Associatively" column on the right gives their evaluation order.

Operator Precedence Chart

Operator Type	**Operator**	**Associatively**
Primary Expression Operators	() [] . -> *expr*++ *expr*--	left-to-right
Unary Operators	* & + - ! ~ ++*expr* --*expr* (*typecast*) sizeof()	right-to-left
	* / %	
	+ -	
	>> <<	
Binary Operators	< > <= >=	left-to-right
	== !=	
	&	
	^	

	|	
	&&	
	||	
Ternary Operator	?:	right-to-left
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= |=	right-to-left
Comma	,	left-to-right

Operators Introduction

An operator is a symbol which helps the user to command the computer to do a certain mathematical or logical manipulations. Operators are used in C language program to operate on data and variables. C has a rich set of operators which can be classified as

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increments and Decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

1. Arithmetic Operators

All the basic arithmetic operations can be carried out in C. All the operators have almost the same meaning as in other languages. Both unary and binary operations are available in C language. Unary operations operate on a singe operand, therefore the number 5 when operated by unary – will have the value –5.

Arithmetic Operators

Operator	Meaning

+	Addition or Unary Plus
–	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulus Operator

Examples of arithmetic operators are

x + y
x - y
-x + y
a * b + c
-a * b

etc.,

here a, b, c, x, y are known as operands. The modulus operator is a special operator in C language which evaluates the remainder of the operands after division.

Example

```
#include //include header file stdio.h
void main() //tell the compiler the start of the program
{
int numb1, num2, sum, sub, mul, div, mod; //declaration of variables
scanf ("%d %d", &num1, &num2); //inputs the operands

sum = num1+num2; //addition of numbers and storing in sum.
printf("\n Thu sum is = %d", sum); //display the output

sub = num1-num2; //subtraction of numbers and storing in sub.
printf("\n Thu difference is = %d", sub); //display the output

mul = num1*num2; //multiplication of numbers and storing in mul.
printf("\n Thu product is = %d", mul); //display the output

div = num1/num2; //division of numbers and storing in div.
printf("\n Thu division is = %d", div); //display the output

mod = num1%num2; //modulus of numbers and storing in mod.
printf("\n Thu modulus is = %d", mod); //display the output
```

```
}
```

Integer Arithmetic

When an arithmetic operation is performed on two whole numbers or integers than such an operation is called as integer arithmetic. It always gives an integer as the result. Let x = 27 and y = 5 be 2 integer numbers. Then the integer operation leads to the following results.

x + y = 32
x – y = 22
x * y = 115
x % y = 2
x / y = 5

In integer division the fractional part is truncated.

Floating point arithmetic

When an arithmetic operation is preformed on two real numbers or fraction numbers such an operation is called floating point arithmetic. The floating point results can be truncated according to the properties requirement. The remainder operator is not applicable for floating point arithmetic operands.

Let x = 14.0 and y = 4.0 then

x + y = 18.0
x – y = 10.0
x * y = 56.0
x / y = 3.50

Mixed mode arithmetic

When one of the operand is real and other is an integer and if the arithmetic operation is carried out on these 2 operands then it is called as mixed mode arithmetic. If any one operand is of real type then the result will always be real thus 15/10.0 = 1.5

2. Relational Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. This is when the relational operator come into picture. C supports the following relational operators.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to

It is required to compare the marks of 2 students, salary of 2 persons, we can compare them using relational operators.

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

exp1 relational operator exp2

Where exp1 and exp2 are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

6.5 <= 25 TRUE
-65 > 0 FALSE
10 < 7 + 5 TRUE

Relational expressions are used in decision making statements of C language such as if, while and for statements to decide the course of action of a running program.

3. Logical Operators

C has the following logical operators, they compare or evaluate logical and relational expressions.

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

Logical AND (&&)

This operator is used to evaluate 2 conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator is true then the whole compound expression is true.

Example

a > b && x = = 10

The expression to the left is a > b and that on the right is x == 10 the whole expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

Logical OR (||)

The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

Example

a < m || a < n

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if a is less than either m or n and when a is less than both m and n.

Logical NOT (!)

The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words it just reverses the value of the expression.

For example

! (x >= y) the NOT expression evaluates to true only if the value of x is neither greater than or equal to y

4. Assignment Operators

The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression.

Example

x = a + b

Here the value of a + b is evaluated and substituted to the variable x.

In addition, C has a set of shorthand assignment operators of the form.

var oper = exp;

Here var is a variable, exp is an expression and oper is a C binary arithmetic operator. The operator oper = is known as shorthand assignment operator

Example

x + = 1 is same as x = x + 1

The commonly used shorthand assignment operators are as follows

Shorthand assignment operators .

Statement with simple assignment operator	Statement with shorthand operator
a = a + 1	a += 1
a = a – 1	a -= 1
a = a * (n+1)	a *= (n+1)

a = a / (n+1)	a /= (n+1)
a = a % b	a %= b

Example for using shorthand assignment operator

```
#define N 100 //creates a variable N with constant value 100
#define A 2 //creates a variable A with constant value 2

main() //start of the program
{
int a; //variable a declaration
a = A; //assigns value 2 to a

while (a < N) //while value of a is less than N
{ //evaluate or do the following
printf("%d \n",a); //print the current value of a
a *= a; //shorthand form of a = a * a
} //end of the loop
} //end of the program
```

Output

2
4
16

5. Increment and Decrement Operators

The increment and decrement operators are one of the unary operators which are very useful in C language. They are extensively used in for and while loops. The syntax of the operators is given below

1. ++ variable name
2. variable name++
3. – –variable name
4. variable name– –

The increment operator ++ adds the value 1 to the current value of operand and the decrement operator – – subtracts the value 1 from the current value of operand.

++variable name and variable name++ mean the same thing when they form 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; (prefix)

In this case the value of y and m would be 6

Suppose if we rewrite the above statement as

m = 5;
y = m++; (post fix)

Then the value of y will be 5 and that of m will be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increments the operand.

6. Conditional or Ternary Operator

The conditional operator consists of 2 symbols the question mark (?) and the colon (:) The syntax for a ternary operator is as follows .

exp1 ? exp2 : exp3

The ternary operator works as follows

exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

For example

a = 10;
b = 15;
x = (a > b) ? a : b

Here x will be assigned to the value of b. The condition follows that the expression is false therefore b is assigned to x.

```
/* Example : to find the maximum value using conditional operator)
#include
void main() //start of the program
{
int i,j,larger; //declaration of variables
printf (“Input 2 integers : ”); //ask the user to input 2 numbers
scanf(“%d %d”,&i, &j); //take the number from standard input and store it
larger = i > j ? i : j; //evaluation using ternary operator
printf(“The largest of two numbers is %d \n”, larger); // print the largest number
} // end of the program
```

Output

Input 2 integers : 34 45
The largest of two numbers is 45

7. Bitwise Operators

C has a distinction of supporting special operators known as bitwise operators for manipulation data at bit level. A bitwise operator operates on each bit of data. Those operators are used for testing, complementing or shifting bits to the right on left. Bitwise operators may not be applied to a float or double.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive
<<	Shift left
>>	Shift right

8. Special Operators

C supports some special operators of interest such as comma operator, size of operator, pointer operators (& and *) and member selection operators (. and ->). The size of and the comma operators are discussed here. The remaining operators are discussed in forth coming chapters.

The Comma Operator

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

For example the statement

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

First assigns 10 to x and 5 to y and finally assigns 15 to value. Since comma has the lowest precedence in operators the parenthesis is necessary. Some examples of comma operator are

In for loops:

for (n=1, m=10, n <=m; n++,m++)

In while loops

While (c=getchar(), c != '10')

Exchanging values.

t = x, x = y, y = t;

The size of Operator

The operator size of gives the size of the data type or variable in terms of bytes occupied in the memory. The operand may be a variable, a constant or a data type qualifier.

Example

```
m = sizeof (sum);
n = sizeof (long int);
k = sizeof (235L);
```

The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.

Example program that employs different kinds of operators. The results of their evaluation are also shown in comparision .

```
main() //start of program
{
int a, b, c, d; //declaration of variables
a = 15; b = 10; c = ++a-b; //assign values to variables
printf (“a = %d, b = %d, c = %d\n”, a,b,c); //print the values
d=b++ + a;
printf (“a = %d, b = %d, d = %d\n, a,b,d);
printf (“a / b = %d\n, a / b);
printf (“a %% b = %d\n, a % b);
printf (“a *= b = %d\n, a *= b);
printf (“%d\n, (c > d) ? 1 : 0 );
printf (“%d\n, (c < d) ? 1 : 0 );
}
```

Notice the way the increment operator ++ works when used in an expression. In the statement c = ++a – b; new value a = 16 is used thus giving value 6 to C. That is a is incremented by 1 before using in expression.

However in the statement d = b++ + a; The old value b = 10 is used in the expression. Here b is incremented after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement.

printf(“a %% b = %d\n”, a%b);

This program also illustrates that the expression

c > d ? 1 : 0

Assumes the value 0 when c is less than d and 1 when c is greater than d.

Type conversions in expressions

Implicit type conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance. This automatic type conversion is known as implicit type conversion.

During evaluation it adheres to very strict rules and type conversion. If the operands are of different types the lower type is automatically converted to the higher type before the

operation proceeds. The result is of higher type.

The following rules apply during evaluating expressions

All short and char are automatically converted to int then

1. If one operand is long double, the other will be converted to long double and result will be long double.

2. If one operand is double, the other will be converted to double and result will be double.

3. If one operand is float, the other will be converted to float and result will be float.

4. If one of the operand is unsigned long int, the other will be converted into unsigned long int and result will be unsigned long int.

5. If one operand is long int and other is unsigned int then .
   a. If unsigned int can be converted to long int, then unsigned int operand will be converted as such and the result will be long int.
   b. Else Both operands will be converted to unsigned long int and the result will be unsigned long int.

6. If one of the operand is long int, the other will be converted to long int and the result will be long int.
7. If one operand is unsigned int the other will be converted to unsigned int and the result will be unsigned int.

Explicit Conversion

Many times there may arise a situation where we want to force a type conversion in a way that is different from automatic conversion.

Consider for example the calculation of number of female and male students in a class

$$\text{Ratio} = \frac{\text{female\_students}}{\text{male\_students}}$$

Since if female_students and male_students are declared as integers, the decimal part will be rounded off and its ratio will represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below.

Ratio = (float) female_students / male_students

The operator float converts the female_students to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed by floating point mode, thus retaining the fractional part of the result. The process of such a local conversion is known as explicit conversion or casting a value. The general form is

(type_name) expression

Specifier Meaning

%c – Print a character
%d – Print a Integer
%i – Print a Integer
%e – Print float value in exponential form.
%f – Print float value
%g – Print using %e or %f whichever is smaller
%o – Print actual value
%s – Print a string
%x – Print a hexadecimal integer (Unsigned) using lower case a – F
%X – Print a hexadecimal integer (Unsigned) using upper case A – F
%a – Print a unsigned integer.
%p – Print a pointer value
%hx – hex short
%lo – octal long
%ld – long unsigned integer.

Input and Output

Input and output are covered in some detail. C allows quite precise control of these. This section discusses input and output from keyboard and screen.

The same mechanisms can be used to read or write data from and to files. It is also possible to treat character strings in a similar way, constructing or analysing them and storing results in variables. These variants of the basic input and output commands are discussed in the next section

---

- The Standard Input Output File
- Character Input / Output
    - getchar
    - putchar
- Formatted Input / Output
    - printf

    - scanf
- Whole Lines of Input and Output
    - gets
    - puts

printf

This offers more structured output than putchar. Its arguments are, in order; a control string, which controls what gets printed, followed by a list of values to be substituted for entries in the control string

Example: int a,b;

printf(" a = %d,b=%d",a,b);.

Control String Entry	What Gets Printed
%d	A Decimal Integer
%f	A Floating Point Value
%c	A Character
%s	A Character String

It is also possible to insert numbers into the control string to control field widths for values to be displayed. For example %6d would print a decimal value in a field 6 spaces wide, %8.2f would print a real value in a field 8 spaces wide with room to show 2 decimal places. Display is left justified by default, but can be right justified by putting a - before the format information, for example %-6d, a decimal integer right justified in a 6 space field

scanf

scanf allows formatted reading of data from the keyboard. Like printf it has a control string, followed by the list of items to be read. However scanf wants to know the address of the items to be read, since it is a function which will change that value. Therefore the names of variables are preceded by the & sign. Character strings are an exception to this. Since a string is already a character pointer, we give the names of string variables unmodified by a leading &.

Control string entries which match values to be read are preceeded by the percentage sign in a similar way to their printf equivalents.

Example: int a,b;

scan f("%d%d",& a,& b);

getchar

getchar returns the next character of keyboard input as an int. If there is an error then EOF (end of file) is returned instead. It is therefore usual to compare this value against EOF before using it. If the return value is stored in a char, it will never be equal to EOF, so error conditions will not be handled correctly.

As an example, here is a program to count the number of characters read until an EOF is encountered. EOF can be generated by typing Control - d.

```
#include <stdio.h>

main()
{   int ch, i = 0;

    while((ch = getchar()) != EOF)
       i ++;

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

putchar

putchar puts its character argument on the standard output (usually the screen).

The following example program converts any typed input into capital letters. To do this it applies the function to upper from the character conversion library c type .h to each character in turn.

```
#include <ctype.h>  /* For definition of toupper */
#include <stdio.h>  /* For definition of getchar, putchar, EOF */

main()
{   char ch;

    while((ch = getchar()) != EOF)
       putchar (toupper(ch));
}
```

gets

gets reads a whole line of input into a string until a new line or EOF is encountered. It is critical to ensure that the string is large enough to hold any expected input lines.

When all input is finished, NULL as defined in studio is returned.

```
#include <stdio.h>

main()
{   char ch[20];
```

```
    gets(x);
    puts(x);
}
```

puts

puts writes a string to the output, and follows it with a new line character.

Example: Program which uses gets and puts to double space typed input.

```
#include <stdio.h>

main()
{       char line[256]; /* Define string sufficiently large to
                           store a line of input */

        while(gets(line) != NULL)   /* Read line        */
        {       puts(line);         /* Print line       */
                printf("\n");       /* Print blank line */
        }
}
```

Note that putchar, printf and puts can be freely used together

Expression Statements

Most of the statements in a C program are *expression statements*. An expression statement is simply an expression followed by a semicolon. The lines

```
        i = 0;
        i = i + 1;
```

and

```
        printf("Hello, world!\n");
```

are all expression statements. (In some languages, such as Pascal, the semicolon separates statements, such that the last statement is not followed by a semicolon. In C, however, the semicolon is a statement terminator; all simple statements are followed by semicolons. The semicolon is also used for a few other things in C; we've already seen that it terminates declarations, too.

UNIT – 3

Branching and looping

CONTROL FLOW STATEMENT

IF- Statement:

It is the basic form where the if statement evaluate a test condition and direct program execution depending on the result of that evaluation.

Syntax:

```
if (Expression)
{
Statement 1;
Statement 2;
}
```

Where a statement may consist of a single statement, a compound statement or nothing as an empty statement. The Expression also referred so as test condition must be enclosed in parenthesis, which cause the expression to be evaluated first, if it evaluate to true (a non zero value), then the statement associated with it will be executed otherwise ignored and the control will pass to the next statement.

Example:

```
if (a>b)
{
printf (“a is larger than b”);
}
```

IF-ELSE Statement:

An if statement may also optionally contain a second statement, the ``else clause," which is to be executed if the condition is not met. Here is an example:

```
if(n > 0)
        average = sum / n;
else    {
        printf("can't compute average\n");
        average = 0;
        }
```

NESTED-IF- Statement:

It's also possible to nest one if statement inside another. (For that matter, it's in general possible to nest any kind of statement or control flow construct within another.) For example, here is a little piece of code which decides roughly which quadrant of the compass you're walking into, based on an x value which is positive if you're walking east, and a y value which is positive if you're walking north:

```
if(x > 0)
        {
        if(y > 0)
                printf("Northeast.\n");
        else    printf("Southeast.\n");
        }
else    {
        if(y > 0)
                printf("Northwest.\n");
        else    printf("Southwest.\n");
        }
```

```
/* Illustates nested if else and multiple arguments to the scanf function.  */
        #include <stdio.h>

        main()
        {
                int    invalid_operator = 0;
                char   operator;
                float  number1, number2, result;

                printf("Enter two numbers and an operator in the format\n");
                printf(" number1 operator number2\n");
                scanf("%f %c %f", &number1, &operator, &number2);

                if(operator == '*')
                        result = number1 * number2;
                else if(operator == '/')
                        result = number1 / number2;
                else if(operator == '+')
                        result = number1 + number2;
                else if(operator == '-')
                        result = number1 - number2;
                else
                        invalid _ operator = 1;

                if( invalid _ operator != 1 )
                printf("%f %c %f is %f\n", number1, operator, number2, result );
                else
                        printf("Invalid operator.\n");
        }
```

Sample Program Output

Enter two numbers and an operator in the format
number1 operator number2
23.2 + 12
23.2 + 12 is 35.2

Switch Case

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

```
estimate(number)
int number;
/* Estimate a number as none, one, two, several, many */
{       switch(number) {
        case 0 :
                printf("None\n");
                break;
        case 1 :
                printf("One\n");
                break;
        case 2 :
                printf("Two\n");
                break;
        case 3 :
        case 4 :
        case 5 :
                printf("Several\n");
                break;
        default :
                printf("Many\n");
                break;
      }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Both if and switch constructs allow the programmer to make a selection from a number of possible actions.

Loops

Looping is a way by which we can execute any some set of statements more than one times continuously .In c there are mainly three types of loops are use :

- while Loop
- do while Loop
- For Loop

While Loop

Loops generally consist of two parts: one or more *control expressions* which (not surprisingly) control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

The general syntax of a while loop is

```
        Initialization

    while( expression )
{
            Statement1
            Statement2
            Statement3

}
```

The most basic *loop* in C is the while loop. A while loop has one control expression, and executes as long as that expression is true. This example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
    int x = 2;

    while(x < 1000)
```

```
{
printf("%d\n", x);
x = x * 2;
}
```

(Once again, we've used braces {} to enclose the group of statements which are to be executed together as the body of the loop.)

For Loop

Our second loop, which we've seen at least one example of already, is the for loop. The general syntax of a while loop is

```
for( Initialization;expression;Increments/decrements )

{
        Statement1
        Statement2
        Statement3

}
```

The first one we saw was:

```
for (i = 0; i < 10; i = i + 1)
        printf ("i is %d\n", i);
```

(Here we see that the for loop has three control expressions. As always, the *statement* can be a brace-enclosed block.)

Do while Loop

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{
  printf("Enter 1 for yes, 0 for no :");
     scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

The break Statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The continue Statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

Take the following example:

```
int i;
for (i=0;i<10;i++)
{

if (i==5)
continue;
printf("%d",i);
if (i==8)
break;
}
```

This code will print 1 to 8 except 5.

Continue means, whatever code that follows the continue statement WITHIN the loop code block will not be exectued and the program will go to the next iteration, in this case, when the program reaches i=5 it checks the condition in the if statement and executes 'continue', everything after continue, which are the printf statement, the next if statement, will not be executed.

Break statement will just stop execution of the look and go to the next statement after

the loop if any. In this case when i=8 the program will jump out of the loop. Meaning, it wont continue till i=9, 10.

Comment:

- The compiler is "line oriented", and parses your program in a line-by-line fashion.
- There are two kinds of comments: single-line and multi-line comments.
- The single-line comment is indicated by "//"

  This means everything after the first occurrence of "//", UP TO THE END OF CURRENT LINE, is ignored.

- The multi-line comment is indicated by the pair "/*" and "*/".

  This means that everything between these two sequences will be ignored. This may ignore any number of lines.

Here is a variant of our first program:

```
/* This is a variant of my first program.
* It is not much, I admit.
*/
int main() {
printf("Hello World!\n");    // that is all?
return(0);
}
```

UNIT – 4

ARRAY AND STRING

Arrays are widely used data type in 'C' language. It is a collection of elements of similar data type. These similar elements could be of all integers, all floats or all characters. An array of character is called as string whereas and array of integer or float is simply called as an array. So array may be defined as a group of elements that share a common name and that are defined by position or index. The elements of an arrays are store in sequential order in memory.
There are mainly two types of Arrays are used:

- One dimensional Array
- Multidimensional Array

One dimensional Array

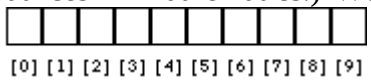
So far, we've been declaring simple variables: the declaration

```
int i;
```

declares a single variable, named i, of type int. It is also possible to declare an *array* of several elements. The declaration

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

declares an array, named a, consisting of ten elements, each of type int. Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a numeric *subscript*. (Arrays in programming are similar to vectors or matrices in mathematics.) We can represent the array a above with a picture like this:



In C, arrays are *zero-based*: the ten elements of a 10-element array are numbered from 0 to 9. The subscript which specifies a single element of an array is simply an integer expression in square brackets. The first element of the array is a[0], the second element is a[1], etc. You can use these ``array subscript expressions" anywhere you can use the name of a simple variable, for example:

```
a[0] = 10;
a[1] = 20;
a[2] = a[0] + a[1];
```

Notice that the subscripted array references (i.e. expressions such as a[0] and a[1]) can appear on either side of the assignment operator. it is possible to initialize some or all elements of an array when the array is defined. The syntax looks like this:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The list of values, enclosed in braces {}, separated by commas, provides the initial values for successive elements of the array.

The subscript does not have to be a constant like 0 or 1; it can be any integral expression. For example, it's common to loop over all elements of an array:

```
int i;

for(i = 0; i < 10; i = i + 1)
        a[i] = 0;
```

This loop sets all ten elements of the array a to 0.

Arrays are a real convenience for many problems, but there is not a lot that C will do with them for you automatically. In particular, you can neither set all elements of an array at once nor assign one array to another; both of the assignments

```
a = 0;                          /* WRONG */
```

and

```
int b[10];
b = a;                          /* WRONG */
```

are illegal.

To set all of the elements of an array to some value, you must do so one by one, as in the loop example above. To copy the contents of one array to another, you must again do so one by one:

```
int b[10];

for(i = 0; i < 10; i = i + 1)
        b[i] = a[i];
```

Remember that for an array declared

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

there is no element a[10]; the topmost element is a[9]. This is one reason that zero-based loops are also common in C. Note that the for loop

```
for(i = 0; i < 10; i = i + 1)
        ...
```

does just what you want in this case: it starts at 0, the number 10 suggests (correctly) that it goes through 10 iterations, but the less-than comparison means that the last trip through the loop has i set to 9. (The comparison i <= 9 would also work, but it would be less clear and therefore poorer style.)

Multidimensional Array

The declaration of an array of arrays looks like this:

```
int a2[5][7];
```

You have to read complicated declarations like these ``inside out." What this one says is that a2 is an array of 5 something's, and that each of the something's is an array of 7 ints. More briefly, ``a2 is an array of 5 arrays of 7 ints," or, ``a2 is an array of array of int." In the declaration of a2, the brackets closest to the identifier a2 tell you what a2 first and foremost is. That's how you know it's an array of 5 arrays of size 7, not the other way around. You can think of a2 as having 5 ``rows" and 7 ``columns," although this interpretation is not mandatory. (You could also treat the ``first" or inner subscript as ``x" and the second as ``y." Unless you're doing something fancy, all you have to worry about is that the subscripts when you access the array match those that you used when you declared it, as in the examples below.)

To illustrate the use of multidimensional arrays, we might fill in the elements of the above array a2 using this piece of code:

```
int i, j;
for(i = 0; i < 5; i = i + 1)
        {
        for(j = 0; j < 7; j = j + 1)

 a2[i][j] = 10 * i + j;
```

```
}
```

This pair of nested loops sets a[1][2] to 12, a[4][1] to 41, etc. Since the first dimension of a2 is 5, the first subscripting index variable, i, runs from 0 to 4. Similarly, the second subscript varies from 0 to 6.

We could print a2 out (in a two-dimensional way, suggesting its structure) with a similar pair of nested loops:

```
for (i = 0; i < 5; i = i + 1)
        {
        for (j = 0; j < 7; j = j + 1)
                printf ("%d\t", a2[i][j]);
        printf ("\n");
        }
```

(The character \t in the printf string is the tab character.)

Just to see more clearly what's going on, we could make the ``row'' and ``column'' subscripts explicit by printing them, too:

```
for(j = 0; j < 7; j = j + 1)
        printf("\t%d:", j);
printf ("\n");

for(i = 0; i < 5; i = i + 1)
        {
        printf("%d:", i);
        for(j = 0; j < 7; j = j + 1)
                printf("\t%d", a2[i][j]);
        printf("\n");
        }
```

This last fragment would print

```
    0:   1:   2:   3:   4:   5:   6:
0:   0    1    2    3    4    5    6
1:  10   11   12   13   14   15   16
2:  20   21   22   23   24   25   26
3:  30   31   32   33   34   35   36
4:  40   41   42   43   44   45   46
```

STRING

String are the combination of number of characters these are used to store any word in any variable of constant. A string is an array of character. It is internally represented in system by using ASCII value. Every single character can have its own ASCII value in the system. A character string is stored in one array of character type.

e.g. “Ram” contains ASCII value per location, when we are using strings and then these strings are always terminated by character ‘\0’. We use conversion specifies %s to set any string we can have any string as follows:-

char nm [25].

When we store any value in nm variable then it can hold only 24 character because at the end of the string one character is consumed automatically by ‘\0’.

#include<string.h>

There are some common inbuilt functions to manipulation on string in string.h file. these are as follows:

1. ***strlen*** *- string length*
2. ***strcp****y - string copy*
3. ***strcm****p - string compare*
4. ***strup****s - string upper*
5. ***strlw****r - string lower*
6. ***strca****t - string concatenate*

UNIT- 5

FUNCTIONS

Function

A function is a ``black box" that we've locked part of our program into. The idea behind a function is that it *compartmentalizes* part of the program, and in particular, that the code within the function has some useful properties:

1. It performs some well-defined task, which will be useful to other parts of the program.
2. It might be useful to other programs as well; that is, we might be able to reuse it (and without having to rewrite it).
3. The rest of the program doesn't have to know the details of how the function is implemented. This can make the rest of the program easier to think about.
4. The function performs its task *well*. It may be written to do a little more than is required by the first program that calls it, with the anticipation that the calling program (or some other program) may later need the extra functionality or improved performance. (It's important that a finished function do its job well, otherwise there might be a reluctance to call it, and it therefore might not achieve the goal of reusability.)
5. By placing the code to perform the useful task into a function, and simply calling the function in the other parts of the program where the task must be performed, the rest of the program becomes clearer: rather than having some large, complicated, difficult-to-understand piece of code repeated wherever the task is being performed, we have a single simple function call, and the name of the function reminds us which task is being performed.
6. Since the rest of the program doesn't have to know the details of how the function is implemented, the rest of the program doesn't care if the function is reimplemented later, in some different way (as long as it continues to perform its same task, of course!). This means that one part of the program can be rewritten, to improve performance or add a new feature (or simply to fix a bug), without having to rewrite the rest of the program.

Functions are probably the most important weapon in our battle against software complexity. You'll want to learn when it's appropriate to break processing out into functions (and also when it's not), and *how* to set up function interfaces to best achieve the qualities mentioned above: reusability, information hiding, clarity, and maintainability.

So what defines a function? It has a *name* that you call it by, and a list of zero or more *arguments* or *parameters* that you hand to it for it to act on or to direct its work; it has a *body* containing the actual instructions (statements) for carrying out the task the

function is supposed to perform; and it may give you back a *return value*, of a particular type.

Here is a very simple function, which accepts one argument, multiplies it by 2, and hands that value back:

```
int multbytwo(int x)
{
        int retval;
        retval = x * 2;
        return retval;
}
```

On the first line we see the return type of the function (int), the name of the function (multbytwo), and a list of the function's arguments, enclosed in parentheses. Each argument has both a name and a type; multbytwo accepts one argument, of type int, named x. The name x is arbitrary, and is used only within the definition of multbytwo. The caller of this function only needs to know that a single argument of type int is expected; the caller does not need to know what name the function will use internally to refer to that argument. (In particular, the caller does not have to pass the value of a variable named x.)

Next we see, surrounded by the familiar braces, the body of the function itself. This function consists of one declaration (of a local variable retval) and two statements. The first statement is a conventional expression statement, which computes and assigns a value to retval, and the second statement is a return statement, which causes the function to return to its caller, and also specifies the value which the function returns to its caller.

The return statement can return the value of any expression, so we don't really need the local retval variable; the function could be collapsed to-

```
int multbytwo(int x)
{
        return x * 2;
}
```

How do we call a function? We've been doing so informally since day one, but now we have a chance to call one that we've written, in full detail. Here is a tiny skeletal program to call multby2:

```
#include <stdio.h>

extern int multbytwo(int);

int main()
{
```

```
        int i, j;
        i = 3;
        j = multbytwo(i);
        printf("%d\n", j);
        return 0;
}
```

This looks much like our other test programs, with the exception of the new line

```
extern int multbytwo(int);
```

This is an *external function prototype declaration*. It is an external declaration, in that it declares something which is defined somewhere else. (We've already seen the defining instance of the function multbytwo, but maybe the compiler hasn't seen it yet.) The function prototype declaration contains the three pieces of information about the function that a caller needs to know: the function's name, return type, and argument type(s). Since we don't care what name the multbytwo function will use to refer to its first argument, we don't need to mention it. (On the other hand, if a function takes several arguments, giving them names in the prototype may make it easier to remember which is which, so names may optionally be used in function prototype declarations.) Finally, to remind us that this is an external declaration and not a defining instance, the prototype is preceded by the keyword extern.

The presence of the function prototype declaration lets the compiler know that we intend to call this function, multbytwo. The information in the prototype lets the compiler generate the correct code for calling the function, and also enables the compiler to check up on our code (by making sure, for example, that we pass the correct number of arguments to each function we call).

Down in the body of main, the action of the function call should be obvious: the line

```
j = multbytwo(i);
```

calls multbytwo, passing it the value of i as its argument. When multbytwo returns, the return value is assigned to the variable j. (Notice that the value of main's local variable i will become the value of multbytwo's parameter x; this is absolutely not a problem, and is a normal sort of affair.)

This example is written out in ``longhand," to make each step equivalent. The variable i isn't really needed, since we could just as well call

```
j = multbytwo(3);
```

And the variable j isn't really needed, either, since we could just as well call

```
printf("%d\n", multbytwo(3));
```

Here, the call to multbytwo is a sub expression which serves as the second argument to printf. The value returned by multbytwo is passed immediately to printf. (Here, as in general, we see the flexibility and generality of expressions in C. An argument passed to a function may be an arbitrarily complex sub expression, and a function call is itself

an expression which may be embedded as a sub expression within arbitrarily complicated surrounding expressions.)

We should say a little more about the mechanism by which an argument is passed down from a caller into a function. Formally, C is *call by value*, which means that a function receives *copies* of the values of its arguments. We can illustrate this with an example. Suppose, in our implementation of multbytwo, we had gotten rid of the unnecessary retval variable like this:

```
int multbytwo(int x)
{
        x = x * 2;
        return x;
}
```

Recursive Functions

A recursive function is one which calls itself. This is another complicated idea which you are unlikely to meet frequently. We shall provide some examples to illustrate recursive functions.

Recursive functions are useful in evaluating certain types of mathematical function. You may also encounter certain dynamic data structures such as linked lists or binary trees. Recursion is a very useful way of creating and accessing these structures.

Here is a recursive version of the Fibonacci function. We saw a non recursive version of this earlier.

```
int fib(int num)
/* Fibonacci value of a number */
{       switch(num) {
        case 0:
                return(0);
                break;
        case 1:
                return(1);
                break;
        default:  /* Including recursive calls */
                return(fib(num - 1) + fib(num - 2));
                break;
        }
}
```

We met another function earlier called power. Here is an alternative recursive version.

```
double power(double val, unsigned pow)
{
      if(pow == 0)  /* pow(x, 0) returns 1 */
            return(1.0);
      else
            return(power(val, pow - 1) * val);
}
```

Notice that each of these definitions incorporate a test. Where an input value gives a trivial result, it is returned directly; otherwise the function calls itself, passing a changed version of the input values. Care must be taken to define functions which will not call themselves indefinitely, otherwise your program will never finish.

The definition of fib is interesting, because it calls itself twice when recursion is used. Consider the effect on program performance of such a function calculating the Fibonacci function of a moderate size number.

Input Value	Number of times `fib` is called
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177

If such a function is to be called many times, it is likely to have an adverse effect on program performance.

Don't be frightened by the apparent complexity of recursion. Recursive functions are sometimes the simplest answer to a calculation. However there is always an alternative non-recursive solution available too. This will normally involve the use of a loop, and may lack the elegance of the recursive solution.

UNIT – 6

Pointer

a pointer is a variable that points to or references a memory location in which data is stored. In the computer, each memory cell has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

Pointer declaration:

A pointer is a variable that contains the memory location of another variable in which data is stored. Using pointer, you start by specifying the type of data stored in the location. The asterisk helps to tell the compiler that you are creating a pointer variable. Finally you have to give the name of the variable. The syntax is as shown below.

```
type * variable name
```

The following example illustrate the declaration of pointer variable :

```
int                                                    *ptr;
float *string;
```

Address operator:

Once we declare a pointer variable then we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

```
ptr=&num;
```

The above code tells that the address where num is stores into the variable ptr. The variable ptr has the value 21260,if num is stored in memory 21260 address then

---

The following program illustrate the pointer declaration :

```
/* A program to illustrate pointer declaration*/

main()
{
int *ptr;
int sum;
sum=45;
ptr=&ptr;
printf (”\n Sum is %d\n”, sum);
printf (”\n The sum pointer is %d”, ptr);
```

```
}
```

Pointer expressions & pointer arithmetic:

In expressions, like other variables pointer variables can be used. For example if p1 and p2 are properly initialized and declared pointers, then the following statements are valid.

```
y=*p1**p2;
sum=sum+*p1;
z= 5* – *p2/p1;
*p2= *p2 + 10;
```

C allows us to subtract integers to or add integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with pointers p1+=; sum+=*p2; etc., By using relational operators, we can also compare pointers like the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

The following program illustrate the pointer expression and pointer arithmetic :

```
/*Program to illustrate the pointer expression and pointer arithmetic*/
#include< stdio.h >
main()
{ int ptr1,ptr2;
int a,b,x,y,z;
a=30;b=6;
ptr1=&a;
ptr2=&b;
x=*ptr1+ *ptr2 6;
y=6*- *ptr1/ *ptr2 +30;
printf(”\nAddress of a +%u”,ptr1);
printf(”\nAddress of b %u”,ptr2);
printf(”\na=%d, b=%d”,a,b);
printf(”\nx=%d,y=%d”,x,y);
ptr1=ptr1 + 70;
ptr2= ptr2;
printf(”\na=%d, b=%d,”a,b);
}
```

Pointers and function:

In a function declaration, the pointer are very much used . Sometimes, only with a pointer a complex function can be easily represented and success. In a function definition, the usage of the pointers may be classified into two groups.

1. **Call by reference**
2. **Call by value.**

Call by value:

We have seen that there will be a link established between the formal and actual parameters when a function is invoked. As soon as temporary storage is created where the value of actual parameters is stored. The formal parameters picks up its value from storage area the mechanism of data transfer between formal and actual parameters allows the actual parameters mechanism of data transfer is referred as call by value. The corresponding formal parameter always represents a local variable in the called function. The current value of the corresponding actual parameter becomes the initial value of formal parameter. In the body of the actual parameter, the value of formal parameter may be changed. In the body of the subprogram, the value of formal parameter may be changed by assignment or input statements. This will not change the value of the actual parameters.

```
/* Include< stdio.h >

void main()
{
int x,y;
x=20;
y=30;
printf(”\n Value of a and b before function call =%d %d”,a,b);
fncn(x,y);
printf(”\n Value of a and b after function call =%d %d”,a,b);
}

fncn(p,q)
int p,q;
{
p=p+p;
q=q+q;
}
```

Call by Reference:

The address should be pointers, when we pass address to a function the parameters receiving. By using pointers, the process of calling a function to pass the address of the variable is known as call by reference. The function which is called by reference can change the value of the variable used in the call.

```
/* example of call by reference*?

/* Include< stdio.h >
void main()
{
int x,y;
x=20;
y=30;
printf("\n Value of a and b before function call =%d %d",a,b);
fncn(&x,&y); printf("\n Value of a and b after function call =%d %d",a,b);
}
fncn(p,q)
int p,q;
{
*p=*p+*p;
*q=*q+*q;
}
```

Pointer to arrays:

an array is actually very much similar like pointer. We can declare as int *a is an address, because a[0] the arrays first element as a[0] and *a is also an address the form of declaration is also equivalent. The difference is pointer can appear on the left of the assignment operator and it is a is a variable that is lvalue. The array name cannot appear as the left side of assignment operator and is constant.

```
/* A program to display the contents of array using pointer*/
main()
{
int a[100];
int i,j,n;
printf("\nEnter the elements of the array\n");
scanf(%d,&n);
printf("Enter the array elements");
for(I=0;I< n;I++)
scanf(%d,&a[I]);
printf("Array element are");
for(ptr=a,ptr< (a+n);ptr++)
printf("Value of a[%d]=%d stored at address %u",j+=,*ptr,ptr);
}
```

Pointers and structures :

We know the name of an array stands for address of its zeros element the same concept applies for names of arrays of structures. Suppose item is an array variable of the struct type. Consider the following declaration:

```
struct products
{
char name[30];
int manufac;
float net;
item[2],*ptr;
```

UNIT - 7

STRUCTURES

What is a Structure?

- Structure is a method of packing the data of different types.
- When we require using a collection of different data items of different data types in that situation we can use a structure.
- A structure is used as a method of handling a group of related data items of different data types.

A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

A structure can be defined as a new named type, thus extending the number of available types. It can use other structures, arrays or pointers as some of its members, though this can get complicated unless you are careful.

Defining a Structure

A structure type is usually defined near to the start of a file using a typedef statement. typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

Here is an example structure definition.

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

This defines a new type student variables of type student can be declared as follows.

```
student st_rec;
```

Notice how similar this is to declaring an int or float.

The variable name is st_rec, it has members called name, course, age and year.

Accessing Members of a Structure

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. To return to the examples above, member name of structure st_rec will behave just like a normal array of char, however we refer to it by the name .

```
st_rec.name
```

Here the dot is an operator which selects a member from a structure.

Where we have a pointer to a structure we could dereference the pointer and then use dot as a member selector. This method is a little clumsy to type. Since selecting a member from a structure pointer happens frequently, it has its own operator -> which acts as follows. Assume that st_ptr is a pointer to a structure of type student We would refer to the name member as.

```
st_ptr -> name
```

```
/* Example program for using a structure*/
#include< stdio.h >
void main()
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}newstudent;
printf(”Enter the student information”);
printf(”Now Enter the student id_no”);
scanf(“%d”,&newstudent.id_no);
```

```
printf(“Enter the name of the student”);
scanf(“%s”,&new student.name);
printf(“Enter the address of the student”);
scanf(“%s”,&new student.address);printf(“Enter the cmbination of the student”);
scanf(“%d”,&new student.combination);printf(Enter the age of the student”);
scanf(“%d”,&new student.age);
printf(“Student information\n”);
printf(“student id_number=%d\n”,newstudent.id_no);
printf(“student name=%s\n”,newstudent.name);
printf(“student Address=%s\n”,newstudent.address);
printf(“students combination=%s\n”,newstudent.combination);
printf(“Age of student=%d\n”,newstudent.age);
}
```

Arrays of structure:

It is possible to define a array of structures for example if we are maintaining information of all the students in the college and if 100 students are studying in the college. We need to use an array than single variables. We can define an array of structures as shown in the following example:

```
structure information
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}
student[100];
```

An array of structures can be assigned initial values just as any other array can. Remember that each element is a structure that must be assigned corresponding initial values as illustrated below.

```
#include< stdio.h >
{
struct info
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
}
struct info std[100];
int I,n;
```

```
printf("Enter the number of students");
scanf("%d",&n);
printf(" Enter Id_no,name address combination age\m");
for(I=0;I < n;I++)
scanf("%d%s%s%s%d",&std[I].id_no,std[I].name,std[I].address,std[I].combination,&s
td[I].age);
printf("\n Student information");
for (I=0;I< n;I++)
printf("%d%s%s%s%d\n",
",std[I].id_no,std[I].name,std[I].address,std[I].combination,std[I].age);
}
```

Structure within a structure:

A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures.

```
struct date
{
int day;
int month;
int year;
};
struct student
{
int id_no;
char name[20];
char address[20];
char combination[3];
int age;
structure date def;
structure date doa;
}oldstudent, newstudent;
```

the sturucture student constains another structure date as its one of its members.

UNIT- 8

UNIONS

Union:

Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computers memory where as each member within a structure is assigned its own unique storage area. Thus unions are used to observe memory. They are useful for

application involving multiple members. Where values need not be assigned to all the members at any one time. Like structures union can be declared using the keyword union as follows:

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

this declares a variable code of type union item. The union contains three members each with a different data type. However we can use only one of them at a time. This is because if only one location is allocated for union variable irrespective of size. The compiler allocates a piece of storage that is large enough to access a union member we can use the same syntax that we use to access structure members. That is

```
code.m
code.p
code.c
```

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

```
code.m=456;
code.p=456.78;
printf("%d",code.m);
```

Would prodece erroneous result..

Enum declarations

There are two kinds of enum type declarations. One kind creates a named type, as in
enum MyEnumType { ALPHA, BETA, GAMMA };
If you give an enum type a name, you can use that type for variables, function arguments and return values, and so on:
enum MyEnumType x; /* legal in both C and C++ */
MyEnumType y; // legal only in C++
The other kind creates an unnamed type. This is used when you want names for constants but don't plan to use the type to declare variables, function arguments, etc. For example, you can write
enum { HOMER, MARGE, BART, LISA, MAGGIE };

Values of enum constants

If you don't specify values for enum constants, the values start at zero and increase by one with each move down the list. For example, given
enum MyEnumType { ALPHA, BETA, GAMMA };
ALPHA has a value of 0, BETA has a value of 1, and GAMMA has a value of 2.

If you want, you may provide explicit values for enum constants, as in

enum FooSize { SMALL = 10, MEDIUM = 100, LARGE = 1000 };

There is an implicit conversion from any enum type to int. Suppose this type exists:

enum MyEnumType { ALPHA, BETA, GAMMA };
Then the following lines are legal:
int i = BETA; // give i a value of 1
int j = 3 + GAMMA; // give j a value of 5

On the other hand, there is *not* an implicit conversion from int to an enum type:

MyEnumType x = 2; // should NOT be allowed by compiler
MyEnumType y = 123; // should NOT be allowed by compiler
Note that it doesn't matter whether the int matches one of the constants of the enum type; the type conversion is always illegal.

Typedefs

A typedef in C is a declaration. Its purpose is to create new types from existing types; whereas a variable declaration creates new memory locations. Since a typedef is a declaration, it can be intermingled with variable declarations, although common practice would be to state typedefs first, then variable declarations. A nice programming convention is to capitalize the first letter of a user-defined type to distinguish it from the built-in types, which all have lower-case names. Also, typedefs are usually global declarations.

Example: Use a Typedef To Create A Synonym for a Type Name

```
typedef int Integer; //Integer can now be used in place of int

int a,b,c,d; //4 variables of type int

Integer e,f,g,h; //the same thing
```

In general, a typedef should never be used to assign a different name to a built-in type name; it just confuses the reader. Usually, a typedef associates a type name with a more complicated type specification, such as an array. A typedef should always be used in situations where the same type definition is used more than once for the same purpose. For example, a vector of 20 elements might represent different aspects of a scientific measurement.

Example: Use a Typedef To Create A Synonym for an Array Type

```
typedef int Vector[20]; //20 integers

Vector a,b;

int a[20], b[20]; //the same thing, but a typedef is preferred
```

Typedefs for Enumerated Types

Every type has constants. For the "int" type, the constants are 1,2,3,4,5; for "char", 'a','b','c'. When a type has constants that have names, like the colors of the rainbow, that type is called an enumerated type. Use an enumerated type for computer representation of common objects that have names like Colors, Playing Cards, Animals, Birds, Fish etc. Enumerated type constants (since they are names) make a program easy to read and understand.

We know that all names in a computer usually are associated with a number. Thus, all of the names (RED, BLUE, GREEN) for an enumerated type are "encoded" with numbers. In eC, if you define an enumerated type, like Color, you cannot add it to an integer; it is not type compatible. In standard C++, anything goes. Also, in eC an enumerated type must always be declared in a typedef before use (in fact, all new types must be declared before use).

Example: Use a Typedef To Create An Enumerated Type

```
typedef enum {RED, BLUE, GREEN} Color;

Color a,b;

a = RED;
a = RED+BLUE; //NOT ALLOWED in eC

if ((a == BLUE) || (a==b)) cout<<"great";
```

Notice that an enumerated type is a code that associates symbols and numbers. The char type can be thought of as an enumeration of character codes. The default code for an enumerated type assigns the first name to the value 0 (RED), second name 1 (BLUE), third 2 (GREEN) etc. The user can, however, override any, or all, of the default codes by specifying alternative values.

UNIT 9

LINKED LIST

linked list is a chain of structs or records called nodes. Each node has at least two members, one of which points to the next item or node in the list! These are defined as *Single Linked Lists* because they only point to the next item, and not the previous. Those that do point to both are called *Doubly Linked Lists* or *Circular Linked Lists*. Please note that there is a distinct difference betweeen Double Linked lists and Circular Linked lists. I won't go into any depth on it because it doesn't concern this tutorial. According to this definition, we could have our record hold **anything** we wanted! The only drawback is that each record must be an instance of the same structure. This means that we couldn't have a record with a char pointing to another structure holding a short, a char array, and a long. Again, they have to be instances of the same structure for this to work. Another cool aspect is that each structure can be located anywhere in memory, each node doesn't have to be linear in memory!

a **linked list** is data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a *link*) to the next record in the sequence.A linked list whose nodes contain two fields: an integer value and a link to the next node.

Linked lists are among the simplest and most common data structures, and are used to implement many important abstract data structures, such as stacks, queues, hash tables, symbolic expressions, skip lists, and many more.

The principal benefit of a linked list over a conventional array is that the order of the linked items may be different from the order that the data items are stored in memory or on disk. For that reason, linked lists allow insertion and removal of nodes at any point in the list, with a constant number of operations.

On the other hand, linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list, or finding a node that contains a given data, or locating the place where a new node should be inserted — may require scanning most of the list elements.

What Linked Lists Look Like

An array allocates memory for all its elements lumped together as one block of memory.
In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using
pointers to connect all its nodes together like the links in a chain.
Each node contains two fields: a "data" field to store whatever element type the list holds
for its client, and a "next" field which is a pointer used to link one node to the next node.
Each node is allocated in the heap with a call to malloc(), so the node memory continues
to exist until it is explicitly deallocated with a call to free(). The front of the list is a
5
pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look
like...
`BuildOneTwoThree()`
This drawing shows the list built in memory by the function BuildOneTwoThree() (the full source code for this function is below). The beginning of the linked list is stored in a
"head" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last
node in the list has its .next field set to NULL to mark the end of the list. Code can access
any node in the list by starting at the head and following the `.next` pointers. Operations
towards the front of the list are fast while operations which access node farther down the
list take longer the further they are from the front. This "linear" cost to access a node is fundamentally more costly then the constant time [ ] access provided by arrays. In this respect, linked lists are definitely less efficient than arrays.
Drawings such as above are important for thinking about pointer code, so most of the examples in this article will associate code with its memory drawing to emphasize the habit. In this case the head pointer is an ordinary local pointer variable, so it is drawn separately on the left to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

The Empty List — NULL

The above is a list pointed to by head is described as being of "length three" since it is made of three nodes with the .next field of the last node set to NULL. There needs to be some representation of the empty list — the list with zero nodes. The most common representation chosen for the empty list is a NULL head pointer. The empty list case is the one common weird "boundary case" for linked list code. All of the code presented in
this article works correctly for the empty list case, but that was not without some effort.

When working on linked list code, it's a good habit to remember to check the empty list case to verify that it works too. Sometimes the empty list case works the same as all the cases, but sometimes it requires some special case code. No matter what, it's a good case
to at least think about.

Linked List Types: Node and Pointer

Before writing the code to build the above list, we need two data types...
• *Node* The type for the nodes which will make up the body of the list. These are allocated in the heap. Each node contains a single client data element and a pointer to the next node in the list. Type: `struct node`

```
struct node {
int data;
struct node* next;
};
```

• *Node Pointer* The type for pointers to nodes. This will be the type of the head pointer and the .next fields inside each node. In C and C++, no separate type declaration is required since the pointer type is just the node type followed by a '*'. Type: `struct node*`

BuildOneTwoThree() Function

Here is simple function which uses pointer operations to build the list {1, 2, 3}. The memory drawing above corresponds to the state of memory at the end of this function. This function demonstrates how calls to malloc() and pointer assignments (=) work to build a pointer structure in the heap.

```
/*
Build the list {1, 2, 3} in the heap and store
its head pointer in a local stack variable.
Returns the head pointer to the caller.
*/
struct node* BuildOneTwoThree() {
struct node* head = NULL;
struct node* second = NULL;
struct node* third = NULL;
head = malloc(sizeof(struct node)); // allocate 3 nodes in the heap
second = malloc(sizeof(struct node));
third = malloc(sizeof(struct node));
head->data = 1; // setup first node
head->next = second; // note: pointer assignment rule
second->data = 2; // setup second node
second->next = third;
third->data = 3; // setup third link
third->next = NULL;
// At this point, the linked list referenced by "head"
// matches the list in the drawing.
return head;


}
```

Basic concepts and nomenclature

Each record of a linked list is often called an **element** or **node**.

The field of each node that contains address of the next node is usually called the ***next* link** or ***next* pointer**. The remaining fields may be called the **data**, **information**, **value**, or **payload** fields.

The **head** of a list is its first node, and the **tail** is the list minus that node (or a pointer thereto). In Lisp and some derived languages, the tail may be called the **CDR** (pronounced *could-R*) of the list, while the payload of the head node may be called the

Linear and circular lists

In the last node of a list, the link field often contains a **null** reference, a special value that is interpreted by programs as meaning "there is no such node". A less common convention is to make it point to the first node of the list; in that case the list is said to be **circular** or **circularly linked**; otherwise it is said to be **open** or **linear**.

Simply-, doubly-, and multiply-linked lists

In a **doubly-linked list**, each node contains, besides the next-node link, a second link field pointing to the *previous* node in the sequence. The two links may be called **forward(s)** and **backwards**. Linked lists that lack such pointers are said to be **simply linked**, or **simple linked lists**.

A doubly-linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

The technique known as XOR-linking allows a doubly-linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.

In a **multiply-linked list**, each node contains two or more link fields, each field being used to connect the same set of data records in a different order (e.g., by name, by department, by date of birth, etc.). (While doubly-linked lists can be seen as special cases of multiply-linked list, the fact that the two orders are opposite to each other leads to simpler and more efficient algorithms, so they are usually treated as a separate case.)

Linked lists vs. arrays

	Array	Linked list
Indexing	$\Theta(1)$	$\Theta(n)$
Inserting / Deleting at end	$\Theta(1)$	$\Theta(1)$ or $\Theta(n)$[1]
Inserting / Deleting in middle (with iterator)	$\Theta(n)$	$\Theta(1)$
Persistent	No	Singly yes
Locality	Great	Poor

Linked lists have several advantages over arrays. Insertion of an element at a specific point of a list is a constant-time operation, whereas insertion in an array may require moving half of the elements, or more. While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", an algorithm that iterates over the elements may have to skip a large number of vacant slots.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while an array will eventually fill up, and then have to be resized — an expensive operation, that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may have to be resized in order to avoid wasting too much space.

On the other hand, arrays allow random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heap sort. Sequential access on arrays is also faster than on linked lists on many machines, because they have greater locality of reference and thus profit more from processor caching.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or Boolean values. It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

Some hybrid solutions try to combine the advantages of the two representations. Unrolled linked lists store several elements in each list node, increasing cache

performance while decreasing memory overhead for references. CDR coding does both these as well, by replacing references with the actual data referenced, which extends off the end of the referencing record.

A good example that highlights the pros and cons of using arrays vs. linked lists is by implementing a program that resolves the Josephus problem. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle *n* times. Once you reach the *n*th person, take them out of the circle and have the members close the circle. Then count around the circle the same *n* times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list vs. an array, because if you view the people as connected nodes in a circular linked list then it shows how easily the linked list is able to delete nodes (as it only has to rearrange the links to the different nodes). However, the linked list will be poor at finding the next person to remove and will need to recurse through the list until it finds that person. An array, on the other hand, will be poor at deleting nodes (or elements) as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the *n*th person in the circle by directly referencing them by their position in the array.

The list ranking problem concerns the efficient conversion of a linked list representation into an array. Although trivial for a conventional computer, solving this problem by a parallel algorithm is complicated and has been the subject of much research.

Simply-linked linear lists vs. other lists

While doubly-linked and/or circular lists have advantages over simply-linked linear lists, linear lists offer some advantages that make them preferable in some situations.

For one thing, a simply-linked linear list is a recursive data structure, because it contains a pointer to a *smaller* object of the same type. For that reason, many operations on simply-linked linear lists (such as merging two lists, or enumerating the elements in reverse order) often have very simple recursive algorithms, much simpler than any solution using iterative commands. While one can adapt those recursive solutions for doubly-linked and circularly-linked lists, the procedures generally need extra arguments and more complicated base cases.

Linear simply-linked lists also allow tail-sharing, the use of a common final portion of sub-list as the terminal portion of two different lists. In particular, if a new node is added at the beginning of a list, the former list remains available as the tail of the new one — a simple example of a persistent data structure. Again, this is not true with the other variants: a node may never belong to two different circular or doubly-linked lists.

In particular, end-sentinel nodes can be shared among simply-linked non-circular lists. One may even use the same end-sentinel node for *every* such list. In Lisp, for example,

every proper list ends with a link to a special node, denoted by `nil` or `()`, whose `CAR` and `CDR` links point to itself. Thus a Lisp procedure can safely take the `CAR` or `CDR` of *any* list.

Indeed, the advantages of the fancy variants are often limited to the complexity of the algorithms, not in their efficiency. A circular list, in particular, can usually be emulated by a linear list together with two variables that point to the first and last nodes, at no extra cost.

Doubly-linked vs. singly-linked

Double-linked lists require more space per node (unless one uses xor-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. To do the same in a singly-linked list, one must have the *previous* node's address. Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.

Circularly-linked vs. linearly-linked

A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. for the corners of a polygon, for a pool of buffers that are used and released in FIFO order, or for a set of processes that should be time-shared in round-robin order. In these applications, a pointer to any node serves as a handle to the whole list.

With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list (e.g., in the implementation of a queue), a circular structure allows one to handle the sructure by a single pointer, instead of two.

A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes nodes in two distinct lists joins the two list into one. This property greatly simplifies some algorithms and data structures, such as the quad-edge and face-edge.

The simplest representation for an empty circular list (when such thing makes sense) has no nodes, and is represented by a null pointer. With this choice, many algorithms have to test for this special case, and handle it separately. By contrast, the use of null to denote an empty linear list is more natural and often creates fewer special cases.

Using sentinel nodes

Sentinel node may simplify certain list operations, by ensuring that the next and/or previous nodes exist for every element, and that even empty lists have at least one node. One may also use a sentinel node at the end of the list, with an appropriate data field, to eliminate some end-of-list tests. For example, when scanning the list looking for a node with a given value *x*, setting the sentinel's data field to *x* makes it unnecessary to test for end-of-list inside the loop. Another example is the merging two sorted lists: if their sentinels have data fields set to +∞, the choice of the next output node does not need special handling for empty lists.

However, sentinel nodes use up extra space (especially in applications that use many short lists), and they may complicate other operations (such as the creation of a new empty list).

However, if the circular list is used merely to simulate a linear list, one may avoid some of this complexity by adding a single sentinel node to every list, between the last and the first data nodes. With this convention, an empty list consists of the sentinel node alone, pointing to itself via the next-node link. The list handle should then be a pointer to the last data node, before the sentinel, if the list is not empty; or to the sentinel itself, if the list is empty.

The same trick can be used to simplify the handling of a doubly-linked linear list, by turning it into a circular doubly-linked list with a single sentinel node. However, in this case, the handle should be a single pointer to the dummy node itself.

Linked list operations

When manipulating linked lists in-place, care must be taken to not use values that you have invalidated in previous assignments. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives pseudocode for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use *null* to refer to an end-of-list marker or sentinel, which may be implemented in a number of ways.

Linearly-linked lists

Singly-linked lists

Our node data structure will have two fields. We also keep a variable *firstNode* which always points to the first node in the list, or is *null* for an empty list.

```
record Node {
    data // The data being stored in the node
```

```
    next // A reference to the next node, null for last
node
 }
 record List {
     Node firstNode    // points to first node of list;
null for empty list
 }
```

Traversal of a singly-linked list is simple, beginning at the first node and following each *next* link until we come to the end:

```
 node := list.firstNode
 while node not null {
     (do something with node.data)
     node := node.next
 }
```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done; instead, you have to locate it while keeping track of the previous node.

```
 function insertAfter(Node node, Node newNode) { // insert
newNode after node
     newNode.next := node.next
     node.next    := newNode
 }
```

Inserting at the beginning of the list requires a separate function. This requires updating *firstNode*.

```
 function insertBeginning(List list, Node newNode) { //
insert node before current first node
     newNode.next   := list.firstNode
     list.firstNode := newNode
 }
```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.

```
 function removeAfter(node node) { // remove node past
this one
     obsoleteNode := node.next
     node.next := node.next.next
```

```
     destroy obsoleteNode
 }
 function removeBeginning(List list) { // remove first
node
     obsoleteNode := list.firstNode
     list.firstNode := list.firstNode.next              //
point past deleted node
     destroy obsoleteNode
 }
```

Notice that removeBeginning() sets *list.firstNode* to *null* when removing the last node in the list.

Since we can't iterate backwards, efficient "insertBefore" or "removeBefore" operations are not possible.

Appending one linked list to another can be inefficient unless a reference to the tail is kept as part of the List structure, because we must traverse the entire first list in order to find the tail, and then append the second list to this. Thus, if two linearly-linked lists are each of length *n*, list appending has asymptotic time complexity of $O(n)$. In the Lisp family of languages, list appending is provided by the `append` procedure.

Many of the special cases of linked list operations can be eliminated by including a dummy element at the front of the list. This ensures that there are no special cases for the beginning of the list and renders both insertBeginning() and removeBeginning() unnecessary. In this case, the first useful data in the list will be found at list.firstNode.next.

Circularly-linked list

In a circularly linked list, all nodes are linked in a continuous circle, without using *null*. For lists with a front and a back (such as a queue), one stores a reference to the last node in the list. The *next* node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Circularly-linked lists can be either singly or doubly linked.

Both types of circularly-linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing *firstNode* and *lastNode*, although if the list may be empty we need a special representation for the empty list, such as a *lastNode* variable which points to some node in the list or is *null* if it's empty; we use such a *lastNode* here. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

circularly-linked lists

Assuming that *someNode* is some node in a non-empty circular simply-linked list, this code iterates through that list starting with *someNode*:

```
function iterate(someNode)
  if someNode  ≠ null
    node := someNode
    do
      do something with node.value
      node := node.next
    while node ≠ someNode
```

Notice that the test "**while** node ≠ someNode" must be at the end of the loop. If it were replaced by the test "" at the beginning of the loop, the procedure would fail whenever the list had only one node.

This function inserts a node "newNode" into a circular linked list after a given node "node". If "node" is null, it assumes that the list is empty.

```
function insertAfter(Node node, Node newNode)
    if node = null
      newNode.next := newNode
    else
      newNode.next := node.next
      node.next := newNode
```

Suppose that "L" is a variable pointing to the last node of a circular linked list (or null if the list is empty). To append "newNode" to the *end* of the list, one may do

```
insertAfter(L, newNode)
L = newNode
```

To insert "newNode" at the *beginning* of the list, one may do

```
insertAfter(L, newNode)
if L = null
  L = newNode
```

Linked lists using arrays of nodes

Languages that do not support any type of reference can still create links by replacing pointers with array indices. The approach is to keep an array of records, where each record has integer fields indicating the index of the next (and possibly previous) node in the array. Not all nodes in the array need be used. If records are not supported as well, parallel arrays can often be used instead.

As an example, consider the following linked list record that uses arrays instead of pointers:

```
record Entry {
    integer next; // index of next entry in array
    integer prev; // previous entry (if double-linked)
    string name;
    real balance;
}
```

By creating an array of these structures, and an integer variable to store the index of the first element, a linked list can be built:

```
integer listHead;
Entry Records[1000];
```

Links between elements are formed by placing the array index of the next (or previous) cell into the Next or Prev field within a given element. For example:

Index	Next	Prev	Name	Balance
0	1	4	Jones, John	123.45
1	-1	0	Smith, Joseph	234.56
2 (listHead)	4	-1	Adams, Adam	0.00
3			Ignore, Ignatius	999.99
4	0	2	Another, Anita	876.54
5				
6				
7				

In the above example, `ListHead` would be set to 2, the location of the first entry in the list. Notice that entry 3 and 5 through 7 are not part of the list. These cells are available for any additions to the list. By creating a `ListFree` integer variable, a free list could be created to keep track of what cells are available. If all entries are in use, the size of the array would have to be increased or some elements would have to be deleted before new entries could be stored in the list.

The following code would traverse the list and display names and account balance:

```
i := listHead;
while i >= 0 { '// loop through the list
     print i, Records[i].name, Records[i].balance // print entry
     i = Records[i].next;
}
```

When faced with a choice, the advantages of this approach include:

- The linked list is relocatable, meaning it can be moved about in memory at will, and it can also be quickly and directly serialized for storage on disk or transfer over a network.
- Especially for a small list, array indexes can occupy significantly less space than a full pointer on many architectures.
- Locality of reference can be improved by keeping the nodes together in memory and by periodically rearranging them, although this can also be done in a general store.
- Naïve dynamic memory allocators can produce an excessive amount of overhead storage for each node allocated; almost no allocation overhead is incurred per node in this approach.
- Seizing an entry from a pre-allocated array is faster than using dynamic memory allocation for each node, since dynamic memory allocation typically requires a search for a free memory block of the desired size.

This approach has one main disadvantage, however: it creates and manages a private memory space for its nodes. This leads to the following issues:

- It increase complexity of the implementation.
- Growing a large array when it is full may be difficult or impossible, whereas finding space for a new linked list node in a large, general memory pool may be easier.
- Adding elements to a dynamic array will occasionally (when it is full) unexpectedly take linear (O (n)) instead of constant time (although it's still an amortized constant).
- Using a general memory pool leaves more memory for other data if the list is smaller than expected or if many nodes are freed.

For these reasons, this approach is mainly used for languages that do not support dynamic memory allocation. These disadvantages are also mitigated if the maximum size of the list is known at the time the array is created.

Language support

Many programming languages such as Lisp and Scheme have singly linked lists built in. In many functional languages, these lists are constructed from nodes, each called a *cons* or *cons cell*. The cons has two fields: the *car*, a reference to the data for that node, and the *cdr*, a reference to the next node. Although cons cells can be used to build other data structures, this is their primary purpose.

In languages that support Abstract data types or templates, linked list ADTs or templates are available for building linked lists. In other languages, linked lists are typically built using references together with records. Here is a complete example in C:

```c
#include <stdio.h>    /* for printf */
#include <stdlib.h>   /* for malloc */

struct node
{
	int data;
	struct node *next; /* pointer to next element in
list */
};

struct node *list_add(struct node **p, int i)
{
	struct node *n = malloc(sizeof(struct node));
	if (n == NULL)
		return NULL;

	n->next = *p; /* the previous element (*p) now
becomes the "next" element */
	*p = n;       /* add new empty element to the front
(head) of the list */
	n->data = i;

	return *p;
}

void list_remove(struct node **p) /* remove head */
{
	if (*p != NULL)
	{
		struct node *n = *p;
		*p = (*p)->next;
		free(n);
	}
}

struct node **list_search(struct node **n, int i)
{
	while (*n != NULL)
	{
		if ((*n)->data == i)
		{
			return n;
		}
		n = &(*n)->next;
	}
	return NULL;
}
```

```
void list_print(struct node *n)
{
	if (n == NULL)
	{
		printf("list is empty\n");
	}
	while (n != NULL)
	{
		printf("print %p %p %d\n", n, n->next, n-
>data);
		n = n->next;
	}
}

int main(void)
{
	struct node *n = NULL;

	list_add(&n, 0); /* list: 0 */
	list_add(&n, 1); /* list: 1 0 */
	list_add(&n, 2); /* list: 2 1 0 */
	list_add(&n, 3); /* list: 3 2 1 0 */
	list_add(&n, 4); /* list: 4 3 2 1 0 */
	list_print(n);
	list_remove(&n);                /* remove first (4)
*/
	list_remove(&n->next);          /* remove new
second (2) */
	list_remove(list_search(&n, 1)); /* remove cell
containing 1 (first) */
	list_remove(&n->next);          /* remove second to
last node (0) */
	list_remove(&n);                /* remove last (3)
*/
	list_print(n);

	return 0;
```

UNIT - 10

FILE MANAGEMENT

What is a File?

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and it sometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

Essentially there are two kinds of files that programmers deal with text files and binary files. These two classes of files will be discussed in the following sections.

ASCII Text files

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time.) A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

Binary files

A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files â€“ they a generally processed using read and write operations simultaneously.

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

Creating a file and output some data

In order to create files we have to learn about File I/O i.e. how to write data into a file and how to read data from a file. We will start this section with an example of writing data to a file. We begin as before with the include statement for stdio.h, then define some variables for use in the example including a rather strange looking new type.

```
/* Program to create a file and write some data the file */
#include <stdio.h>
#include <stdio.h>
main( )
{
    FILE *fp;
    char stuff[25];
    int index;
    fp = fopen("TENLINES.TXT","w"); /* open for writing */
    strcpy(stuff,"This is an example line.");
    for (index = 1; index <= 10; index++)
            fprintf(fp,"%s Line number %d\n", stuff, index);
    fclose(fp); /* close the file before ending program */
}
```

The type FILE is used for a file variable and is defined in the stdio.h file. It is used to define a file pointer for use in file operations. Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the file name is. We do this with the fopen() function illustrated in the first line of the program. The file pointer, fp in our case, points to the file and two arguments are required in the parentheses, the file name first, followed by the file type.

The file name is any valid DOS file name, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you donâ€™t have a file by this name, that is good because we will create one and put some data into it. You are permitted to include a directory with the file name. The directory must, of course, be a valid directory otherwise an error will occur. Also, because of the way C handles literal strings, the directory separation character â€˜\â€™ must be written twice. For example, if the file is to be stored in the \PROJECTS sub directory then the file name should be entered as "\\PROJECTS\\TENLINES.TXT". The second parameter is the file attribute and can be any of three letters, r, w, or a, and must be lower case.

Reading (r)

When an r is used, the file is opened for reading, a w is used to indicate a file to be used for writing, and an indicates that you desire to append additional data to the data already in an existing file. Most C compilers have other file attributes available; check your Reference Manual for details. Using the r indicates that the file is assumed to be a text file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

Here is a small program that reads a file and display its contents on screen. /* Program to display the contents of a file on screen */

```
#include <stdio.h>
void main()
{
  FILE *fopen(), *fp;
  int c;
  fp = fopen("prog.c","r");
  c = getc(fp) ;
  while (c!= EOF)
  {
                    putchar(c);
                    c = getc(fp);
  }
  fclose(fp);
}
```

Writing (w)

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

Here is the program to create a file and write some data into the file.

```
#include <stdio.h>
int main()
{
 FILE *fp;
 file = fopen("file.txt","w");
 /*Create a file and add text*/
 fprintf(fp,"%s","This is just an example :)"); /*writes data to the file*/
 fclose(fp); /*done!*/
 return 0;
}
```

Appending (a):

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be positioned at the end of the present data so that any new data will be added to any data that already exists in the file. Using the a indicates that the file is assumed to be a text file.

Here is a program that will add text to a file which already exists and there is some text in the file.

```
#include <stdio.h>
int main()
{
   FILE *fp
   file = fopen("file.txt","a");
   fprintf(fp,"%s","This is just an example :)"); /*append some text*/
   fclose(fp);
   return 0;
}
```

Outputting to the file

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, fprintf replaces our familiar printf function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the printf statement.

Closing a file

To close a file you simply use the function fclose with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file

because the system will close all open files before returning to DOS, but it is good programming practice for you to close all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program.

You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to. Compile and run this program. When you run it, you will not get any output to the monitor because it doesnâ€™t generate any. After running it, look at your directory for a file named TENLINES.TXT and type it; that is where your output will be. Compare the output with that specified in the program; they should agree! Do not erase the file named TENLINES.TXT yet; we will use it in some of the other examples in this section.

Reading from a text file

Now for our first program that reads from a file. This program begins with the familiar include, some data definitions, and the file opening statement which should require no explanation except for the fact that an r is used here because we want to read it.

```
#include <stdio.h>
  main( )
  {
    FILE *fp;
    char c;
    funny = fopen("TENLINES.TXT", "r");
    if (fp == NULL)
                        printf("File doesn't exist\n");
    else {
     do {
      c = getc(fp); /* get one character from the file
      */
        putchar(c); /* display it on the monitor
      */
      } while (c != EOF); /* repeat until EOF (end of file)
    */
     }
   fclose(fp);
  }
```

In this program we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn’t, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we can test. The main body of the program is one do while loop in which a single character is read from the file and

output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated. At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the getc function is a character, so we can use a char variable for this purpose. There is a problem that could develop here if we happened to use an unsigned char however, because C usually returns a minus one for an EOF - which an unsigned char type variable is not capable of containing. An unsigned char type variable can only have the values of zero to 255, so it will return a 255 for a minus one in C. This is a very frustrating problem to try to find. The program can never find the EOF and will therefore never terminate the loop. This is easy to prevent: always have a char or int type variable for use in returning an EOF. There is another problem with this program but we will worry about it when we get to the next program and solve it with the one following that.

After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of TENLINES.TXT and run the program again to see that the NULL test actually works as stated. Be sure to change the name back because we are still not finished with TENLINES.TXT.

UNIT 11

C - PREPROCESSOR

Overview

The C preprocessor, often known as *cpp*, is a *macro processor* that is used automatically by the C compiler to transform your program before compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor is intended to be used only with C, C++, and Objective-C source code. In the past, it has been abused as a general text processor. It will choke on input which does not obey C's lexical rules. For example, apostrophes will be interpreted as the beginning of character constants, and cause errors. Also, you cannot rely on it preserving characteristics of the input which are not significant to C-family languages. If a Makefile is preprocessed, all the hard tabs will be removed, and the Makefile will not work.

Having said that, you can often get away with using cpp on things which are not C. Other Algol-ish programming languages are often safe (Pascal, Ada, etc.) So is assembly, with caution. -traditional-cpp mode preserves more white space, and is otherwise more permissive. Many of the problems can be avoided by writing C or C++ style comments instead of native language comments, and keeping macros simple

Include Syntax

Both user and system header files are included using the preprocessing directive `#include'. It has two variants:

`#include <`*file*`>`
: This variant is used for system header files. It searches for a file named *file* in a standard list of system directories. You can prepend directories to this list with the -I option (see Invocation).

`#include "`*file*`"`
: This variant is used for header files of your own program. It searches for a file named *file* first in the directory containing the current file, then in the quote directories and then the same directories used for <*file*>. You can prepend directories to the list of quote directories with the -iquote option.

The argument of `#include', whether delimited with quote marks or angle brackets, behaves like a string constant in that comments are not recognized, and macro names are not expanded. Thus, `#include <x/*y>` specifies inclusion of a system header file named `x/*y`.

However, if backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, `#include "x\n\\y"` specifies a filename containing three backslashes. (Some systems interpret `\' as a pathname separator. All of these also interpret `/' the same way. It is most portable to use only `/'.)

It is an error if there is anything (other than comments) on the line after the file name.

Object-like Macros

An *object-like macro* is a simple identifier which will be replaced by a code fragment. It is called object-like because it looks like a data object in code that uses it. They are most commonly used to give symbolic names to numeric constants.

You create macros with the `#define' directive. `#define' is followed by the name of the macro and then the token sequence it should be an abbreviation for, which is variously referred to as the macro's *body*, *expansion* or *replacement list*. For example,

```
#define BUFFER_SIZE 1024
```

defines a macro named `BUFFER_SIZE` as an abbreviation for the token `1024`. If somewhere after this `#define' directive there comes a C statement of the form .

```
foo = (char *) malloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and *expand* the macro `BUFFER_SIZE`. The C compiler will see the same tokens as it would if you had written .

```
foo = (char *) malloc (1024);
```

By convention, macro names are written in uppercase. Programs are easier to read when it is possible to tell at a glance which names are macros.

The macro's body ends at the end of the `#define' line. You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will all come out on one line. For example,

```
#define NUMBERS 1, \
                2, \
                3
int x[] = { NUMBERS };
     ==> int x[] = { 1, 2, 3 };
```

The most common visible consequence of this is surprising line numbers in error messages.

There is no restriction on what can go in a macro body provided it decomposes into valid preprocessing tokens. Parentheses need not balance, and the body need not resemble valid C code. (If it does not, you may get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially. Macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces

```
foo = X;
bar = 4;
```

When the preprocessor expands a macro name, the macro's expansion replaces the macro invocation, then the expansion is examined for more macros to expand. For example,

```
#define TABLESIZE BUFSIZE
#define BUFSIZE 1024
TABLESIZE
     ==> BUFSIZE
     ==> 1024
```

`TABLESIZE` is expanded first to produce `BUFSIZE`, then that macro is expanded to produce the final result, `1024`.

Notice that `BUFSIZE` was not defined when `TABLESIZE` was defined. The `#define' for `TABLESIZE` uses exactly the expansion you specify—in this case, `BUFSIZE`—and does not check to see whether it too contains macro names. Only when you *use* `TABLESIZE` is the result of its expansion scanned for more macro names.

This makes a difference if you change the definition of `BUFSIZE` at some point in the source file. `TABLESIZE`, defined as shown, will always expand using the definition of `BUFSIZE` that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Conditional Syntax

A conditional in the C preprocessor begins with a *conditional directive*: `#if', `#ifdef' or `#ifndef'.

- Ifdef
- If
- Defined
- Else
- Elif

Ifdef

The simplest sort of conditional is

```
#ifdef MACRO

controlled text

#endif /* MACRO */
```

This block is called a *conditional group*. *controlled text* will be included in the output of the preprocessor if and only if *MACRO* is defined. We say that the conditional *succeeds* if *MACRO* is defined, *fails* if it is not.

The *controlled text* inside of a conditional can include preprocessing directives. They are executed only if the conditional succeeds. You can nest conditional groups inside other conditional groups, but they must be completely nested. In other words, `#endif' always matches the nearest `#ifdef' (or `#ifndef', or `#if'). Also, you cannot start a conditional group in one file and end it in another.

Even if a conditional fails, the *controlled text* inside it is still run through initial transformations and tokenization. Therefore, it must all be lexically valid C. Normally the only way this matters is that all comments and string literals inside a failing conditional group must still be properly ended.

The comment following the `#endif' is not required, but it is a good practice if there is a lot of *controlled text*, because it helps people match the `#endif' to the corresponding `#ifdef'. Older programs sometimes put *MACRO* directly after the `#endif' without enclosing it in a comment. This is invalid code according to the C standard. CPP accepts it with a warning. It never affects which `#ifndef' the `#endif' matches.

Sometimes you wish to use some code if a macro is *not* defined. You can do this by writing `#ifndef' instead of `#ifdef'. One common use of `#ifndef' is to include code only the first time a header file is included. See Once-Only Headers.

If

The `#if' directive allows you to test the value of an arithmetic expression, rather than the mere existence of one macro. Its syntax is

```
#if expression

controlled text

#endif /* expression */
```

expression is a C expression of integer type, subject to stringent restrictions. It may contain

- Integer constants.
- Character constants, which are interpreted as they would be in normal code.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and logical operations (`&&` and `||`). The latter two obey the usual short-circuiting rules of standard C.
- Macros. All macros in the expression are expanded before actual computation of the expression's value begins.
- Uses of the `defined` operator, which lets you check whether macros are defined in the middle of an `#if'.
- Identifiers that are not macros, which are all considered to be the number zero. This allows you to write `#if MACRO` instead of `#ifdef MACRO`, if you know that MACRO, when defined, will always have a nonzero value. Function-like macros used without their function call parentheses are also treated as zero.

Defined

The special operator `defined` is used in `#if' and `#elif' expressions to test whether a certain name is defined as a macro. `defined` *name* and `defined (`*name*`)` are both expressions whose value is 1 if *name* is defined as a macro at the current point in the program, and 0 otherwise. Thus, `#if defined MACRO` is precisely equivalent to `#ifdef MACRO`.

`defined` is useful when you wish to test more than one macro for existence at once. For example,

```
#if defined (__vax__) || defined (__ns16000__)
```

would succeed if either of the names `__vax__` or `__ns16000__` is defined as a macro.

Conditionals written like this:

```
#if defined BUFSIZE && BUFSIZE >= 1024
```

can generally be simplified to just `#if BUFSIZE >= 1024`, since if `BUFSIZE` is not defined, it will be interpreted as having the value zero.

If the `defined` operator appears as a result of a macro expansion, the C standard says the behavior is undefined. GNU cpp treats it as a genuine `defined` operator and evaluates it normally. It will warn wherever your code uses this feature if you use the command-line option `-pedantic`, since other compilers may handle it differently.

Else

The `#else' directive can be added to a conditional to provide alternative text to be used if the condition fails. This is what it looks like:

```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```

If *expression* is nonzero, the *text-if-true* is included and the *text-if-false* is skipped. If *expression* is zero, the opposite happens.

You can use `#else' with `#ifdef' and `#ifndef', too.

Elif

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have

```
#if X == 1
...
#else /* X != 1 */
#if X == 2
...
#else /* X != 2 */
...
#endif /* X != 2 */
#endif /* X != 1 */
```

Another conditional directive, `#elif', allows this to be abbreviated as follows:

```
#if X == 1
 ...
 #elif X == 2
 ...
 #else /* X != 2 and X != 1*/
```

```
...
#endif /* X != 2 and X != 1*/
```

`#elif' stands for “else if”. Like `#else', it goes in the middle of a conditional group and subdivides it; it does not require a matching `#endif' of its own. Like `#if', the `#elif' directive includes an expression to be tested. The text following the `#elif' is processed only if the original `#if'-condition failed and the `#elif' condition succeeds.

More than one `#elif' can go in the same conditional group. Then the text after each `#elif' is processed only if the `#elif' condition succeeds after the original `#if' and all previous `#elif' directives within it have failed.

`#else' is allowed after any number of `#elif' directives, but `#elif' may not follow `#else'.