Educature

LECTURE NOTE

on

PROGRAMMING IN “C”

By

Educature **“The Online Education Portal”**

SYLLABUS

Module –I

C Language Fundamentals.
Character set, Identifiers, keyword, data types, Constants and variables, statements, expression, operators, precedence of operators, Input-output, Assignments, control structures decision making and branching.

Module -II

Arrays, Functions and Strings: Declaration, manipulation and String – handling functions, monolithic vs. Modular programs, user defined vs. standard functions, formal vs. actual arguments, function – category, function prototypes, parameter passing, recursion, and storage classes: auto, extern, global, static.

Module –III

Pointers, Structures, Unions, File handling:

Pointer variable and its importance, pointer arithmetic, passing parameters, Declaration of structures, pointer to pointer, pointer to structure, pointer to function, union, dynamic memory allocation, file managements.

CONTENTS

Lecture 13: String library functions

Lecture 14: functions, categories

Lecture 15: functions categories cont..

Lecture 16: Actual arguments and Formal arguments, call by value call by reference

Lecture 17:local, global, static variable

Lecture 18: monolithic vs modular programming, Storage classes

Lecture 19:storage class cont.., pointer

Lecture 20: pointer comparison, increment decrement

Lecture 21: precedence level of pointer, pointer comparison

Lecture 22: pointer to pointer, pointer to structure

Lecture 23: pointer initialization, accessing elements

Module: 3

Lecture 24: size of Structure in, array vs structure, array within structure

Lecture 25: passing structure to function, Nested Structure

Lecture 26: Union

Lecture 27: nesting of unions, dynamic memory allocation

Lecture 28: dynamic memory allocation conti…

Lecture 29: dynamic array, file

Lecture 30: file operation

Lecture 31: file operation on string

Lecture Note: 1

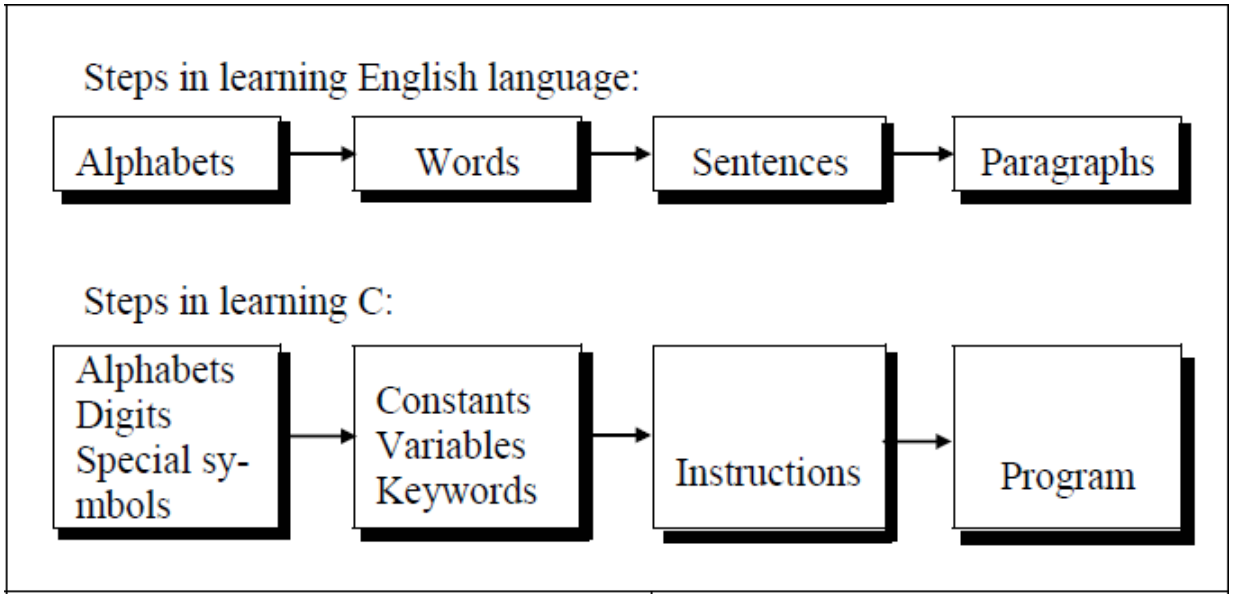
Introduction to **C**

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc

ANSI C standard emerged in the early 1980s, this book was split into two titles: The original was still called ***Programming in C***, and the title that covered ANSI C was called ***Programming in ANSI C***. This was done because it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. It was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C. Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is **reliable**, **simple** and **easy** to use. often heard today is – "C has been already superceded by languages like C++, C# and Java.

Program

There is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an **instruction**. A group of instructions would be combined later on to form a **program**. So



a computer ***program*** is just a collection of the instructions necessary to solve a specific problem. The basic operations of a computer system form what is known as the computer's ***instruction set.*** And the approach or method that is used to solve the problem is known as an ***algorithm***.

So for as programming language concern these are of two types.

1) Low level language

2) High level language

Low level language:

Low level languages are **machine level** and **assembly level language**. In machine level language computer only understand digital numbers i.e. in the form of 0 and 1. So, instruction given to the computer is in the form binary digit, which is difficult to implement instruction in binary code. This type of program is not portable, difficult to maintain and also error prone. The **assembly language** is on other hand modified version of machine level language. Where instructions are given in English like word as ADD, SUM, MOV etc. It is easy to write and understand but not understand by the machine. So the translator used here is assembler to translate into machine level. Although language is bit easier, programmer has to know low level details related to low level language. In the assembly level language the data are stored in the computer register, which varies for different computer. Hence it is not portable.

High level language:

These languages are machine independent, means it is portable. The language in this category is Pascal, Cobol, Fortran etc. High level languages are understood by the machine. So it need to translate by the translator into machine level. A translator is software which is used to translate high level language as well as low level language in to machine level language.

Three types of translator are there:

Compiler

Interpreter

Assembler

Compiler and interpreter are used to convert the high level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Both compiler and interpreter perform the same task but there working is different. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas interpreter read only one line of the source code and convert it to object code. If it check error, statement by statement and hence of take more time.

Integrated Development Environments (IDE)

The process of editing, compiling, running, and debugging programs is often managed by a single integrated application known as an Integrated Development Environment, or IDE for short. An IDE is a windows-based program that allows us to easily manage large software programs, edit files in windows, and compile, link, run, and debug programs.

On Mac OS X, CodeWarrior and Xcode are two IDEs that are used by many programmers. Under Windows, Microsoft Visual Studio is a good example of a popular IDE. Kylix is a popular IDE for developing applications under Linux. Most IDEs also support program development in several different programming languages in addition to C, such as C# and C++.

Lecture Note: 2

Structure of C Language program

1) Comment line

2) Preprocessor directive

3) Global variable declaration

4) main function( )

{

Local variables;

Statements;

}

User defined function

}

}

Comment line

It indicates the purpose of the program. It is represented as

/*……..*/

Comment line is used for increasing the readability of the program. It is useful in explaining the program and generally used for documentation. It is enclosed within the decimeters. Comment line can be single or multiple line but should not be nested. It can be anywhere in the program except inside string constant & character constant.

Preprocessor Directive:

#include<stdio.h> tells the compiler to include information about the standard input/output library. It is also used in symbolic constant such as #define PI 3.14(value). The stdio.h (standard input output header file) contains definition &declaration of system defined function such as printf( ), scanf( ), pow( ) etc. Generally printf() function used to display and scanf() function used to read value

Global Declaration:

This is the section where variable are declared globally so that it can be access by all the functions used in the program. And it is generally declared outside the function :

main()

It is the user defined function and every function has one main() function from where actually program is started and it is encloses within the pair of curly braces.

The main( ) function can be anywhere in the program but in general practice it is placed in the first position.

Syntax :

```
  main()
{
……..
……..
……..
}
```

The main( ) function return value when it declared by data type as

```
 int main( )
{
return 0
```

```
}
```

The main function does not return any value when void (means null/empty) as

```
void main(void ) or void main()
{
 printf (“C language”);
}
```

Output: C language

The program execution start with opening braces and end with closing brace.

And in between the two braces declaration part as well as executable part is mentioned. And at the end of each line, the semi-colon is given which indicates statement termination.

/*First c program with return statement*/

```
#include <stdio.h>
int main (void)
{
printf ("welcome to c Programming language.\n");
return 0;
}
```

Output: welcome to c programming language.

Steps for Compiling and executing the Programs

A compiler is a software program that analyzes a program developed in a particular computer language and then translates it into a form that is suitable for execution

on a particular computer system. Figure below shows the steps that are involved in entering, compiling, and executing a

computer program developed in the C programming language and the typical Unix commands that would be entered from the command line.

Step 1: The program that is to be compiled is first typed into a *file* on the computer system. There are various conventions that are used for naming files, typically be any name provided the last two characters are **".c"** or file with extension .c. So, the file name **prog1.c** might be a valid filename for a C program. A text editor is usually used to enter the C program into a file. For example, vi is a popular text editor used on Unix systems. The program that is entered into the file is known as the ***source program*** because it represents the original form of the program expressed in the C language.

Step 2: After the source program has been entered into a file, then proceed to have it compiled. The compilation process is initiated by typing a special command on the system. When this command is entered, the name of the file that contains the source program must also be specified. For example, under Unix, the command to initiate program compilation is called **cc**. If we are using the popular GNU C compiler, the command we use is **gcc**.

Typing the line

gcc prog1.c or cc prog1.c

In the first step of the compilation process, the compiler examines each program

statement contained in the source program and checks it to ensure that it conforms to the syntax and semantics of the language. If any mistakes are discovered by the compiler during this phase, they are reported to the user and the compilation process ends right there. The errors then have to be corrected in the source program (with the use of an editor), and the compilation process must be restarted. Typical errors reported during this phase of compilation might be due to an expression that has unbalanced parentheses (**syntactic error**), or due to the use of a variable that is not "defined" (**semantic error**).

Step 3: When all the syntactic and semantic errors have been removed from the program, the compiler then proceeds to take each statement of the program and translate it into a "lower" form that is equivalent to assembly language program needed to perform the identical task.

Step 4: After the program has been translated the next step in the compilation process is to translate the assembly language statements into actual machine instructions. The assembler takes each assembly language statement and converts it into a binary format known as ***object code***, which is then written into another file on the system. This file has the same name as the source file under Unix, with the last letter an "**o**" (**for *object***) instead of a "**c**".

Step 5: After the program has been translated into object code, it is ready to be ***linked.*** This process is once again performed automatically whenever the cc or gcc command is issued under Unix. The purpose of the linking phase is to get the program into a final form for execution on the computer.

If the program uses other programs that were previously processed by the compiler, then during this phase the programs are linked together. Programs that are used from the system's program ***library*** are also searched and linked together with the object program during this phase.

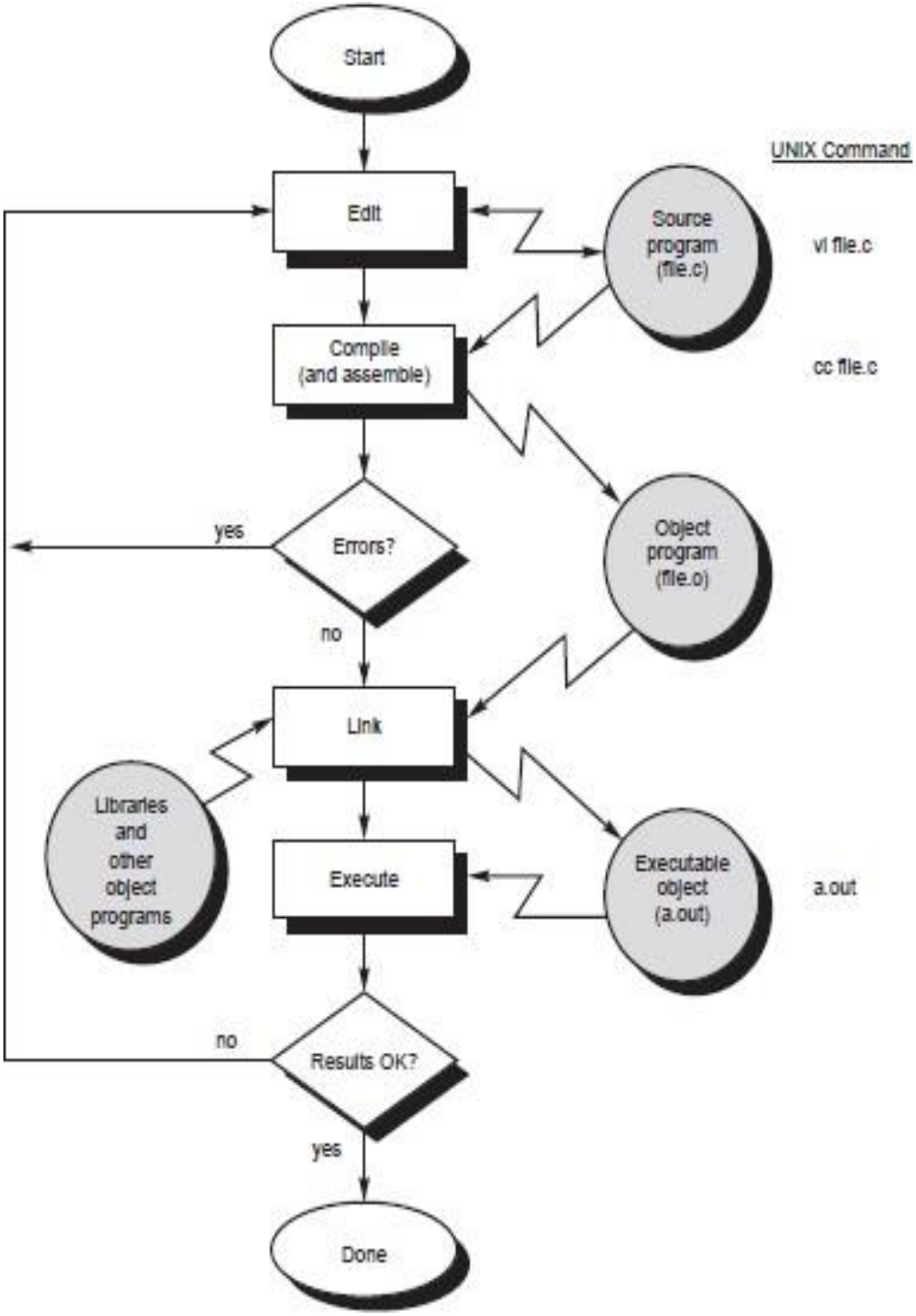
The process of compiling and linking a program is often called ***building*.**

The final linked file, which is in an executable ***object*** code format, is stored in another file on the system, ready to be run or ***executed*.** Under Unix, this file is called **a.out** by default. Under Windows, the executable file usually has the same name as the source file, with the c extension replaced by an exe extension.

Step 6: To subsequently execute the program, the command **a.out** has the effect of *loading* the program called **a.out** into the computer's memory and initiating its execution.

When the program is executed, each of the statements of the program is sequentially executed in turn. If the program requests any data from the user, known as ***input***, the program temporarily suspends its execution so that the input can be entered. Or, the program might simply wait for an ***event***, such as a mouse being clicked, to occur. Results that are displayed by the program, known as ***output,*** appear in a window, sometimes called the ***console***. If the program does not produce the desired results, it is necessary to go back and reanalyze the program's logic. This is known as the ***debugging phase***, during which an attempt is made to remove all the known problems or ***bugs*** from the program. To do this, it will most

likely be necessary to make changes to original source program.



/* Simple program to add two numbers ..... */

```
#include <stdio.h>
int main (void)
{
int  v1, v2, sum;                    //v1,v2,sum are variables and int is data type declared
v1 = 150;
v2 = 25;
sum = v1 + v2;
printf ("The sum of %i and %i is= %i\n", v1, v2, sum);
return 0;
}
```

Output:

The sum of 150 and 25 is=175

Lectu re Note: 3

Character set

A character denotes any alphabet, digit or special symbol used to represent information. Valid alphabets, numbers and special symbols allowed in C are

Alphabets	A, B, ….., Y, Z a, b, ……, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

Identifiers

Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc. Rules for naming identifiers are:

1) name should only consists of alphabets (both upper and lower case), digits and underscore (_) sign.

2) first characters should be alphabet or underscore

3) name should not be a keyword

4) since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.

5) identifiers are generally given in some meaningful name such as value, net_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognize 31 characters. Some invalid identifiers are 5cb, int, res#, avg no etc.

Keyword

There are certain words reserved for doing specific task, these words are known as **reserved word** or **keywords.** These words are predefined and always written in lower case or small letter. These keywords cann't be used as a variable name as it assigned with fixed meaning. Some examples are **int, short, signed, unsigned, default, volatile, float, long, double, break, continue, typedef, static, do, for, union, return, while, do, extern, register, enum, case, goto, struct, char, auto, const etc.**

data types

Data types refer to an extensive system used for declaring variables or functions of different types before its use. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time.

C has the following 4 types of data types

basic built-in data types: int, float, double, char

Enumeration data type: enum

Derived data type: pointer, array, structure, union

Void data type: void

A variable declared to be of type int can be used to contain integral values only—that is, values that do not contain decimal places. A variable declared to be of type float can be used for storing floating- point numbers (values containing decimal places). The double type is the same as type float, only with roughly twice the precision. The char data type can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon similarly A variable declared char can only store character type value.

There are two types of type qualifier in c

Size qualifier: short, long

Sign qualifier: signed, unsigned

When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default sign qualifier is assumed. The range of values for signed data types is less than that of unsigned data type. Because in signed type, the left most bit is used to represent sign, while in unsigned type this bit is also used to represent the value. The size and range of the different data types on a 16 bit machine is given below:

Basic data type	Data type with type qualifier	Size (byte)	Range
char	char or signed char Unsigned char	1 1	-128 to 127 0 to 255
int	int or signed int unsigned int short int or signed short int unsigned short int long int or signed long int unsigned long int	2 2 1 1 4 4	-32768 to 32767 0 to 65535 -128 to 127 0 to 255 -2147483648 to 2147483647 0 to 4294967295
float	float	4	-3.4E-38 to 3.4E+38
double	double Long double	8 10	1.7E-308 to 1.7E+308 3.4E-4932 to 1.1E+4932

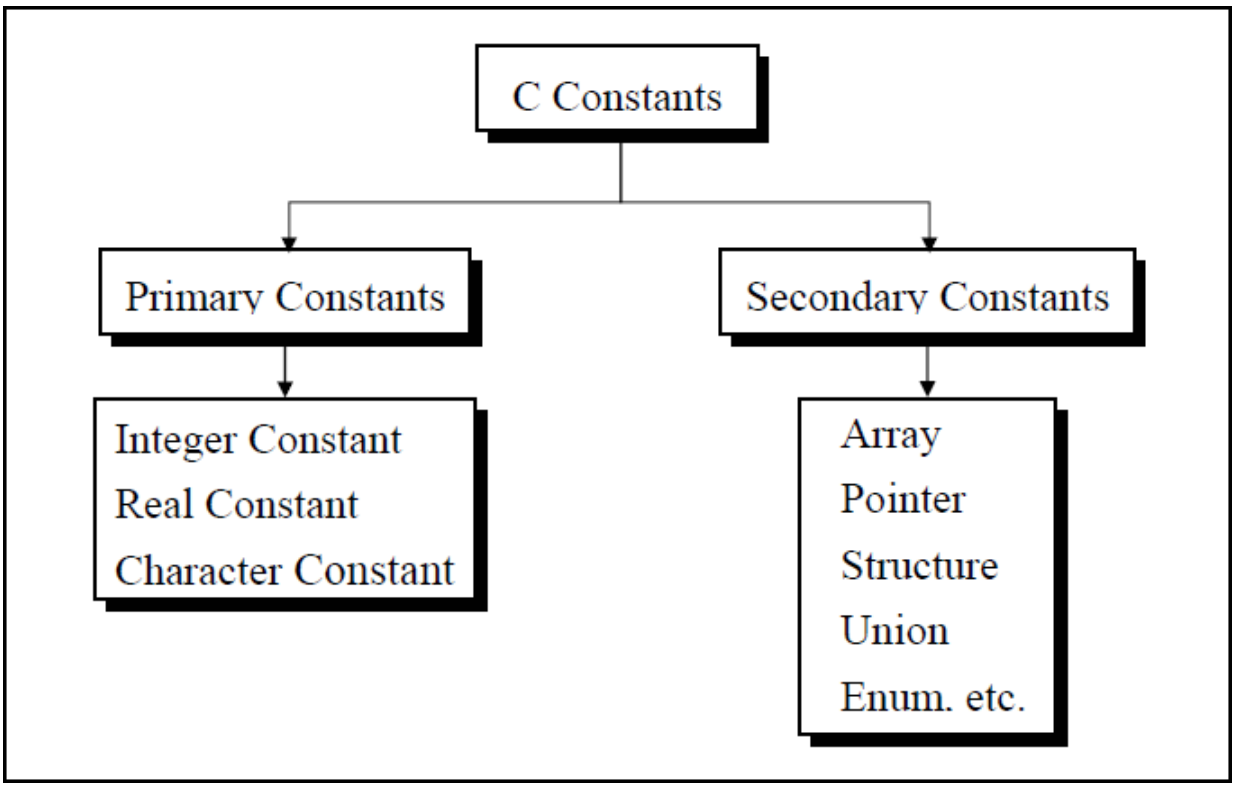
Lecture Note: 4

Constants

Constant is a any value that cannot be changed during program execution. In C, any number, single character, or character string is known as a *constant*. A constant is an entity that doesn't change whereas a variable is an entity that may change. For example, the number 50 represents a constant integer value. The character string "Programming in C is fun.\n" is an example of a constant character string. C constants can be divided into two major categories:

Primary Constants
Secondary Constants

These constants are further categorized as



Numeric constant
Character constant
String constant

Numeric constant: Numeric constant consists of digits. It required minimum size of 2 bytes and max 4 bytes. It may be positive or negative but by default sign is always positive. No comma or space is allowed within the numeric constant and it must have at least 1 digit. The allowable range for integer constants is -32768 to 32767. Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be even greater. Mean by a 16-bit or a 32-bit compiler, what range of an Integer constant has to do with the type of compiler.

It is categorized a **integer constant** and **real constant.** An integer constants are whole number which have no decimal point. Types of integer constants are:

Decimal constant: 0 ------9(base 10)
Octal constant: 0 ------ 7(base 8)
Hexa decimal constant: 0----9, A----- F(base 16)

In decimal constant first digit should not be zero unlike octal constant first digit must be zero(as 076, 0127) and in hexadecimal constant first two digit should be 0x/ 0X (such as 0x24, 0x87A). By default type of integer constant is integer but if the value of integer constant is exceeds range then value represented by integer type is taken to be unsigned integer or long integer. It can also be explicitly mention integer and unsigned integer type by suffix l/L and u/U.

Real constant is also called floating point constant. To construct real constant we must follow the rule of ,

-real constant must have at least one digit.
-It must have a decimal point.
-It could be either positive or negative.
-Default sign is positive.
-No commas or blanks are allowed within a real constant. Ex.: +325.34
426.0
-32.76

To express small/large real constant exponent(scientific) form is used where number is written in mantissa and exponent form separated by e/E. Exponent can be positive or negative integer but mantissa can be real/integer type, for example $3.6*10^5=3.6e+5$. By default type of floating point constant is double, it can also be explicitly defined it by suffix of f/F.

Character constant

Character constant represented as a single character enclosed within a single quote. These can be single digit, single special symbol or white spaces such as '9','c','$', ' ' etc. Every character constant has a unique integer like value in machine's character code as if machine using ASCII (American standard code for information interchange). Some numeric value associated with each upper and lower case alphabets and decimal integers are as:

A ----------Z ASCII value (65-90)

a ----------z ASCII value (97-122)

0----------9 ASCII value (48-59)

; ASCII value (59)

String constant

Set of characters are called string and when sequence of characters are enclosed within a double quote (it may be combination of all kind of symbols) is a string constant. String constant has zero, one or more than one character and at the end of the string null character(\0) is automatically placed by compiler. Some examples are ",sarathina" , "908", "3"," ", "A" etc. In C although same characters are enclosed within single and double quotes it represents different meaning such as "A" and 'A' are different because first one is string attached with null character at the end but second one is character constant with its corresponding ASCII value is 65.

Symbolic constant

Symbolic constant is a name that substitute for a sequence of characters and, characters may be numeric, character or string constant. These constant are generally defined at the beginning of the program as

#define name value , here name generally written in upper case for example

```
#define MAX 10

#define CH 'b'

 #define NAME "sony"
```

Variables

Variable is a data name which is used to store some data value or symbolic names for storing program computations and results. The value of the variable can be change during the execution. The rule for naming the variables is same as the naming identifier. Before used in the program it must be declared. Declaration of variables specify its name, data types and range of the value that variables can store depends upon its data types.

Syntax:

```
 int a;

char c;

float f;
```

Variable initialization

When we assign any initial value to variable during the declaration, is called initialization of variables. When variable is declared but contain undefined value then it is called garbage value. The variable is initialized with the assignment operator such as

```
   Data type variable name=constant;

Example: int a=20;

            Or int a;

                a=20;
```

statements

Lecture Note: 5

Expressions

An expression is a combination of variables, constants, operators and function call. It can be arithmetic, logical and relational for example:-

```
int z=  x+y    // arithmatic expression

 a>b      //relational

a==b        // logical
func(a, b)   // function call
```

Expressions consisting entirely of constant values are called *constant expressions*. So, the expression
121 + 17 - 110
is a constant expression because each of the terms of the expression is a constant value. But if i were declared to be an integer variable, the expression
180 + 2 – j
would not represent a constant expression.

Operator

This is a symbol use to perform some operation on variables, operands or with the constant. Some operator required 2 operand to perform operation or Some required single operation.

Several operators are there those are, arithmetic operator, assignment, increment , decrement, logical, conditional, comma, size of , bitwise and others.

1. Arithmatic Operator

This operator used for numeric calculation. These are of either Unary arithmetic operator, Binary arithmetic operator. Where Unary arithmetic operator required

only one operand such as +,-, ++, --,!, tiled. And these operators are addition, subtraction, multiplication, division. Binary arithmetic operator on other hand required two operand and its operators are +(addition), -(subtraction), *(multiplication), /(division), %(modulus). But modulus cannot applied with floating point operand as well as there are no exponent operator in c.

Unary (+) and Unary (-) is different from addition and subtraction.

When both the operand are integer then it is called integer arithmetic and the result is always integer. When both the operand are floating point then it is called floating arithmetic and when operand is of integer and floating point then it is called mix type or mixed mode arithmetic . And the result is in float type.

2.Assignment Operator

A value can be stored in a variable with the use of assignment operator. The assignment operator(=) is used in assignment statement and assignment expression. Operand on the left hand side should be variable and the operand on the right hand side should be variable or constant or any expression. When variable on the left hand side is occur on the right hand side then we can avoid by writing the compound statement. For example,

```
int x= y;
int  Sum=x+y+z;
```

3.Increment and Decrement

The Unary operator ++, --, is used as increment and decrement which acts upon single operand. Increment operator increases the value of variable by one .Similarly decrement operator decrease the value of the variable by one. And these operator can only used with the variable, but cann't use with expression and constant as ++6 or ++(x+y+z).

It again categories into prefix post fix . In the prefix the value of the variable is incremented 1st, then the new value is used, where as in postfix the operator is written after the operand(such as m++,m--).

EXAMPLE

let y=12;

z= ++y;

y= y+1;

z= y;

Similarly in the postfix increment and decrement operator is used in the operation . And then increment and decrement is perform.

EXAMPLE

let x= 5;

y= x++;

y=x;

x= x+1;

4.Relational Operator

It is use to compared value of two expressions depending on their relation. Expression that contain relational operator is called relational expression.

Here the value is assign according to true or false value.

a.(a>=b) || (b>20)

b.(b>a) && (e>b)

c. 0(b!=7)

5. Conditional Operator

It sometimes called as ternary operator. Since it required three expressions as operand and it is represented as (? , :).

SYNTAX

exp1 ? exp2 :exp3

Here exp1 is first evaluated. It is true then value return will be exp2 . If false then exp3.

EXAMPLE

```
void main()
{
 int a=10, b=2
  int s= (a>b) ? a:b;
  printf("value is:%d");
 }
```

Output:

Value is:10

6. Comma Operator

Comma operator is use to permit different expression to be appear in a situation where only one expression would be used. All the expression are separator by comma and are evaluated from left to right.

EXAMPLE

```
int i, j, k, l;
for(i=1,j=2;i<=5;j<=10;i++;j++)
```

7. Sizeof Operator

Size of operator is a Unary operator, which gives size of operand in terms of byte that occupied in the memory. An operand may be variable, constant or data type qualifier.

Generally it is used make portable program(program that can be run on different machine) . It determines the length of entities, arrays and structures when their size are not known to the programmer. It is also use to allocate size of memory dynamically during execution of the program.

EXAMPLE

```
main( )

{

int sum;

float f;

printf( "%d%d" ,size of(f), size of (sum) );

printf("%d%d", size of(235 L), size of(A));

}
```

Lecture Note: 6

8. Bitwise Operator

Bitwise operator permit programmer to access and manipulate of data at bit level. Various bitwise operator enlisted are

one's complement	(~)
bitwise AND	(&)
bitwise OR	(|)
bitwise XOR	(^)
left shift	(<<)
right shift	(>>)

These operator can operate on integer and character value but not on float and double. In bitwise operator the function showbits( ) function is used to display the binary representation of any integer or character value.

In one's complement all 0 changes to 1 and all 1 changes to 0. In the bitwise OR its value would obtaining by 0 to 2 bits.

As the bitwise OR operator is used to set on a particular bit in a number. Bitwise AND the logical AND.

It operate on 2operands and operands are compared on bit by bit basic. And hence both the operands are of same type.

Logical or Boolean Operator

Operator used with one or more operand and return either value zero (for false) or one (for true). The operand may be constant, variables or expressions. And the expression that combines two or more expressions is termed as logical expression. C has three logical operators :

Operator	Meaning
&&	AND
||	OR
!	NOT

Where logical NOT is a unary operator and other two are binary operator. Logical AND gives result true if both the conditions are true, otherwise result is false. And logial OR gives result false if both the condition false, otherwise result is true.

Precedence and associativity of operators

Operators	Description	Precedence level	Associativity
()	function call	**1**	left to right
[]	array subscript		
→	arrow operator		
.	dot operator		
+	unary plus	2	right to left
-	unary minus		
++	increment		
- -	decrement		
!	logical not		
~	1's complement		
*****	indirection		
&	address		
(data type)	type cast		
sizeof	size in byte		
*	multiplication	3	left to right
/	division		
%	modulus		
+	addition	4	left to right

-	subtraction		
<<	left shift	**5**	left to right
>>	right shift		
<=	less than equal to	6	left to right
>=	greater than equal to		
<	less than		
>	greater than		
==	equal to	7	left to right
!=	not equal to		
&	bitwise AND	8	left to right
^	bitwise XOR	9	left to right
|	bitwise OR	10	left to right
&&	logical AND	11	
||	logical OR	12	
?:	conditional operator	13	
=, *=, /=, %= &=, ^=, <<= >>=	assignment operator	14	right to left
,	comma operator	15	

Lecture Note: 7

Control Statement

Generally C program statement is executed in a order in which they appear in the program. But sometimes we use decision making condition for execution only a part of program, that is called control statement. Control statement defined

how the control is transferred from one part to the other part of the program. There are several control statement like if...else, switch, while, do. ..while, for loop, break, continue, goto etc.

Loops in C

Loop:-it is a block of statement that performs set of instructions. In loops

Repeating particular portion of the program either a specified number of time or until a particular no of condition is being satisfied.

There are three types of loops in c

1.While loop

2.do while loop

3.for loop

While loop

Syntax:-

```
while(condition)
{
Statement 1;
Statement 2;
}
Or      while(test condition)
        Statement;
```

The test condition may be any expression .when we want to do something a fixed no of times but not known about the number of iteration, in a program then while loop is used.

Here first condition is checked if, it is true body of the loop is executed else, If condition is false control will be come out of loop.

Example:-

```
/* wap to print 5 times welcome to C" */
#include<stdio.h>
void main()
{
int p=1;
While(p<=5)
{
printf("Welcome to C\n");
P=p+1;
}
}
```

Output: Welcome to C

Welcome to C

Welcome to C

Welcome to C

So as long as condition remains true statements within the body of while loop will get executed repeatedly.

do while loop

This (do while loop) statement is also used for looping. The body of this loop may contain single statement or block of statement. The syntax for writing this statement is:

Syntax:-

```
Do
{
Statement;
}
while(condition);
```

Example:-

```
#include<stdio.h>
void main()
{
int X=4;
 do
{
 Printf("%d",X);
X=X+1;
```

```
  }whie(X<=10);
    Printf(“ ”);
}
```

Output: 4 5 6 7 8 9 10

Here firstly statement inside body is executed then condition is checked. If the condition is true again body of loop is executed and this process continue until the condition becomes false. Unlike while loop semicolon is placed at the end of while.

There is minor difference between while and do while loop, while loop test the condition before executing any of the statement of loop. Whereas do while loop test condition after having executed the statement at least one within the loop.

If initial condition is false while loop would not executed it’s statement on other hand do while loop executed it’s statement at least once even If condition fails for first time. It means do while loop always executes at least once. **Notes:**

Do while loop used rarely when we want to execute a loop at least once.

Lecture Note: 8

for loop

In a program, for loop is generally used when number of iteration are known in advance. The body of the loop can be single statement or multiple statements. Its syntax for writing is:

Syntax:-

```
for(exp1;exp2;exp3)
{
Statement;
}
```

Or

```
for(initialized counter; test counter; update counter)
{
Statement;
}
```

Here exp1 is an initialization expression, exp2 is test expression or condition and exp3 is an update expression. Expression 1 is executed only once when loop started and used to initialize the loop variables. Condition expression generally uses relational and logical operators. And updation part executed only when after body of the loop is executed.

Example:-

```
void main()
{
int i;
for(i=1;i<10;i++)
{
```

```
 Printf(“ %d ”, i);
  }
}
```

Output:-1 2 3 4 5 6 7 8 9

Nesting of loop

When a loop written inside the body of another loop then, it is known as nesting of loop. Any type of loop can be nested in any type such as while, do while, for. For example nesting of for loop can be represented as :

```
void main()
{
int i,j;
for(i=0;i<2;i++)
for(j=0;j<5; j++)
printf(“%d %d”, i, j);
 }
```

Output: i=0

j=0 1 2 3 4

i=1

j=0 1 2 3 4

Break statement(break)

Sometimes it becomes necessary to come out of the loop even before loop condition becomes false then break statement is used. Break statement is used inside loop and switch statements. It cause immediate exit from that loop in which it appears and it is generally written with condition. It is written with the keyword as **break.** When break statement is encountered loop is terminated and control is transferred to the statement, immediately after loop or situation where we want to jump out of the loop instantly without waiting to get back to conditional state.

When break is encountered inside any loop, control automatically passes to the first statement after the loop. This break statement is usually associated with **if** statement.

Example :

```
void main()
{
int j=0;
for(;j<6;j++)
if(j==4)
break;
}
```

Output:

0 1 2 3

Continue statement (key word continue)

Continue statement is used for continuing next iteration of loop after skipping some statement of loop. When it encountered control automatically passes through the beginning of the loop. It is usually associated with the if statement. It is useful when we want to continue the program without executing any part of the program.

The difference between break and continue is, when the break encountered loop is terminated and it transfer to the next statement and when continue is encounter control come back to the beginning position.

In while and do while loop after continue statement control transfer to the test condition and then loop continue where as in, for loop after continue control transferred to the updating expression and condition is tested.

Example:-

```
void main()

{

int n;

for(n=2; n<=9; n++)

{

if(n==4)

continue;

printf("%d", n);

  }

}

Printf("out of loop");

}
```

Output: 2 3 5 6 7 8 9 out of loop

Lecture Note: 9

if statement

Statement execute set of command like when condition is true and its syntax is

```
If (condition)
Statement;
```

The statement is executed only when condition is true. If the if statement body is consists of several statement then better to use pair of curly braces. Here in case condition is false then compiler skip the line within the if block.

```
void main()
{
    int n;
     printf (" enter a number:");
      scanf("%d",&n);
          If (n>10)
        Printf(" number is grater");
      }
```

Output:

```
Enter a number:12
Number is greater
```

if…..else ... Statement

it is bidirectional conditional control statement that contains one condition & two possible action. Condition may be true or false, where non-zero value regarded as true & zero value regarded as false. If condition are satisfy true, then a single or block of statement executed otherwise another single or block of statement is executed.

Its syntax is:-

```
if (condition)
 {
 Statement1;
  Statement2;
 }
    else
       {
        Statement1;
        Statement2;
         }
```

Else statement cannot be used without if or no multiple else statement are allowed within one if statement. It means there must be a if statement with in an else statement.

Example:-

/* To check a number is eve or odd */

```
 void main()
{
  int n;
      printf (“enter a number:”);
     sacnf (“%d”, &n);
      If (n%2==0)
         printf (“even number”);
else
         printf(“odd number”);
}
```

Output: enter a number:121

odd number

Lecture Note: 10

Nesting of if …else

When there are another if else statement in if-block or else-block, then it is called nesting of if-else statement.

Syntax is :-

```
if (condition)
 {
```

```
If (condition)
    Statement1;
else
    statement2;
}
    Statement3;
```

If….else LADDER

In this type of nesting there is an if else statement in every else part except the last part. If condition is false control pass to block where condition is again checked with its if statement.

Syntax is :-

```
if (condition)
    Statement1;
else if (condition)
    statement2;
else if (condition)
    statement3;
else
    statement4;
```

This process continue until there is no if statement in the last block. if one of the condition is satisfy the condition other nested “else if” would not executed.

But it has disadvantage over if else statement that, in if else statement whenever the condition is true, other condition are not checked. While in this case, all condition are checked.

Lecture Note: 11

ARRAY

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string. Each data item of an array is called an element. And each element is unique and located in separated memory location. Each of elements of an array share a variable but each element having different index no. known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. One dimensional array is known as vector and two dimensional arrays are known as matrix.

ADVANTAGES: array variable can store more than one value at a time where other variable can store one value at a time.

Example:

```
int  arr[100];
```

int mark[100];

DECLARATION OF AN ARRAY :

Its syntax is :

Data type array name [size];

int arr[100];

int mark[100];

int a[5]={10,20,30,100,5}

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte for each element etc. The size of the array operates the number of elements that can be stored in an array and it may be a int constant or constant int expression.

We can represent individual array as :

int ar[5];

ar[0], ar[1], ar[2], ar[3], ar[4];

Symbolic constant can also be used to specify the size of the array as:

#define SIZE 10;

INITIALIZATION OF AN ARRAY:

After declaration element of local array has garbage value. If it is global or static array then it will be automatically initialize with zero. An explicitly it can be initialize that

Data type array name [size] = {value1, value2, value3…}

Example:

in ar[5]={20,60,90, 100,120}

Array subscript always start from zero which is known as lower bound and upper value is known as upper bound and the last subscript value is one less than the size of array. Subscript can be an expression i.e. integer value. It can be any integer, integer constant, integer variable, integer expression or return value from functional call that yield integer value.

So if i & j are not variable then the valid subscript are

ar [i*7],ar[i*i],ar[i++],ar[3];

The array elements are standing in continuous memory locations and the amount of storage required for hold the element depend in its size & type.

Total size in byte for 1D array is:

Total bytes=size of (data type) * size of array.

Example : if an array declared is:

int [20];

Total byte= 2 * 20 =40 byte.

ACCESSING OF ARRAY ELEMENT:

/*Write a program to input values into an array and display them*/

```
#include<stdio.h>

int main()

{

int arr[5],i;

for(i=0;i<5;i++)

{

printf("enter a value for arr[%d] \n",i);

scanf("%d",&arr[i]);

}
```

```
printf("the array elements are: \n");
for (i=0;i<5;i++)
{
printf("%d\t",arr[i]);
}
return 0;
}
```

OUTPUT:

Enter a value for arr[0] = 12

Enter a value for arr[1] =45

Enter a value for arr[2] =59

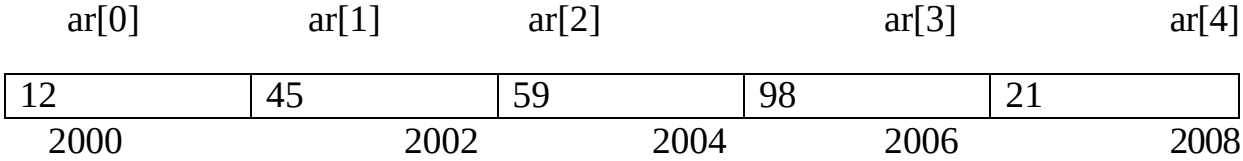
Enter a value for arr[3] =98

Enter a value for arr[4] =21

The array elements are 12 45 59 98 21

Example: From the above example value stored in an array are and occupy its memory addresses 2000, 2002, 2004, 2006, 2008 respectively.

a[0]=12, a[1]=45, a[2]=59, a[3]=98, a[4]=21



Example 2:

```
/* Write a program to add 10 array elements */
#include<stdio.h>
void main()
{
int i ;
int arr [10];
int sum=o;
for (i=0; i<=9; i++)
{
printf (“enter the %d element \n”, i+1);
scanf (“%d”, &arr[i]);
}
for (i=0; i<=9; i++)
{
sum = sum + a[i];
}
printf (“the sum of 10 array elements is %d”, sum);
}
```

OUTPUT:

Enter a value for arr[0] =5

Enter a value for arr[1] =10

Enter a value for arr[2] =15

Enter a value for arr[3] =20

Enter a value for arr[4] =25

Enter a value for arr[5] =30

Enter a value for arr[6] =35

Enter a value for arr[7] =40

Enter a value for arr[8] =45

Enter a value for arr[9] =50

Sum = 275

while initializing a single dimensional array, it is optional to specify the size of array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

For example:-

```
int  marks[]={99,78,50,45,67,89};
```

If during the initialization of the number the initializers is less then size of array, then all the remaining elements of array are assigned value zero .

For example:-

```
int marks[5]={99,78};
```

Here the size of the array is 5 while there are only two initializers so After this initialization, the value of the rest elements are automatically occupied by zeros such as

Marks[0]=99 , Marks[1]=78 , Marks[2]=0, Marks[3]=0, Marks[4]=0

Again if we initialize an array like

int array[100]={0};

Then the all the element of the array will be initialized to zero. If the number of initializers is more than the size given in brackets then the compiler will show an error.

For example:-

```
int arr[5]={1,2,3,4,5,6,7,8};//error
```

we cannot copy all the elements of an array to another array by simply assigning it to the other array like, by initializing or declaring as

```
int a[5] ={1,2,3,4,5};

int b[5];

b=a;//not valid
```

(**note**:-here we will have to copy all the elements of array one by one, using for loop.)

Single dimensional arrays and functions

```
/*program to pass array elements to a function*/

#include<stdio.h>

void main()

{

int arr[10],i;

printf("enter the array elements\n");

for(i=0;i<10;i++)

{

scanf("%d",&arr[i]);

check(arr[i]);

}

}
```

```
void check(int num)
{
if(num%2=0)
{
printf("%d is even \n",num);
}
else
{
printf("%d is odd \n",num);
}
}
```

Lecture Note: 12

Two dimensional arrays

Two dimensional array is known as matrix. The array declaration in both the array i.e.in single dimensional array single subscript is used and in two dimensional array two subscripts are is used.

Its syntax is

Data-type array name[row][column];

Or we can say 2-d array is a collection of 1-D array placed one below the other.

Total no. of elements in 2-D array is calculated as **row*column**

Example:-

int a[2][3];

Total no of elements=row*column is 2*3 =6

It means the matrix consist of 2 rows and 3 columns

For example:-

20 2 7

8 3 15

Positions of 2-D array elements in an array are as below

00 01 02

10 11 12

a [0][0] a [0][0] a [0][0] a [0][0] a [0][0] a [0][0]

20	2	7	8	3	15
2000	2002	2004	2006	2008	

Accessing 2-d array /processing 2-d arrays

For processing 2-d array, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

For example

int a[4][5];

for reading value:-

```
for(i=0;i<4;i++)
{
    for(j=0;j<5;j++)
    {
        scanf("%d",&a[i][j]);
    }
}
```

For displaying value:-

```
for(i=0;i<4;i++)
{
for(j=0;j<5;j++)
    {
        printf("%d",a[i][j]);
    }
}
```

Initialization of 2-d array:

2-D array can be initialized in a way similar to that of 1-D array. for example:-

int mat[4][3]={11,12,13,14,15,16,17,18,19,20,21,22};

These values are assigned to the elements row wise, so the values of elements after this initialization are

Mat[0][0]=11, Mat[1][0]=14, Mat[2][0]=17 Mat[3][0]=20

Mat[0][1]=12, Mat[1][1]=15, Mat[2][1]=18 Mat[3][1]=21

Mat[0][2]=13, Mat[1][2]=16, Mat[2][2]=19 Mat[3][2]=22

While initializing we can group the elements row wise using inner braces.

for example:-

```
int mat[4][3]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};
```

And while initializing , it is necessary to mention the 2nd dimension where 1st dimension is optional.

int mat[][3];

int mat[2][3];

int mat[][];

int mat[2][]; invalid

If we **initialize an array** as

```
int mat[4][3]={{11},{12,13},{14,15,16},{17}};
```

Then the compiler will assume its all rest value as 0,which are not defined.

Mat[0][0]=11, Mat[1][0]=12, Mat[2][0]=14, Mat[3][0]=17

Mat[0][1]=0, Mat[1][1]=13, Mat[2][1]=15 Mat[3][1]=0

Mat[0][2]=0, Mat[1][2]=0, Mat[2][2]=16, Mat[3][2]=0

In memory map whether it is 1-D or 2-D, elements are stored in one contiguous manner.

We can also give the size of the 2-D array by using symbolic constant

Such as

```
#define ROW 2;
```

```
#define COLUMN 3;

int mat[ROW][COLUMN];
```

String

Array of character is called a string. It is always terminated by the NULL character. String is a one dimensional array of character.

We can initialize the string as

char name[]={‘j’,’o’,’h’,’n’,’\o’};

Here each character occupies 1 byte of memory and last character is always NULL character. Where ’\o’ and 0 (zero) are not same, where **ASCII** value of ‘\o’ is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation.

From the above we can represent as;

J	o	h	N	‘\o’

The terminating NULL is important because it is only the way that the function that work with string can know, where string end.

String can also be **initialized** as;

char name[]=”John”;

Here the NULL character is not necessary and the compiler will assume it automatically.

String constant (string literal)

A string constant is a set of character that enclosed within the double quotes and is also called a literal. Whenever a string constant is written anywhere in a program it is stored somewhere in a memory as an array of characters terminated by a NULL character (‘\o’).

Example – “m”

“Tajmahal”

“My age is %d and height is %f\n”

The string constant itself becomes a pointer to the first character in array.

Example-char crr[20]=”Taj mahal”;

1000	1001	1002	1003	1004	1005	1006	1007	100	1009
T	a	j		M	A	H	a	l	\o

It is called base address.

Lecture Note: 13

String library function

There are several string library functions used to manipulate string and the prototypes for these functions are in header file “string.h”. Several string functions are

strlen()

This function return the length of the string. i.e. the number of characters in the string excluding the terminating NULL character.

It accepts a single argument which is pointer to the first character of the string.

For example-

strlen(“suresh”);

It return the value 6.

In array version to calculate legnth:-

```
int str(char str[])
{
    int i=0;
    while(str[i]!='\o')
    {
        i++;
    }
    return i;
}
```

Example:-

```
#include<stdio.h>
#include<string.h>
void main()
{
    char str[50];
    print("Enter a string:");
```

```
    gets(str);
    printf("Length of the string is %d\n",strlen(str));
}
```

Output:

Enter a string: C in Depth

Length of the string is 8

strcmp()

This function is used to compare two strings. If the two string match, strcmp() return a value 0 otherwise it return a non-zero value. It compare the strings character by character and the comparison stops when the end of the string is reached or the corresponding characters in the two string are not same.

strcmp(s1,s2)

return a value:

<0 when s1<s2

=0 when s1=s2

>0 when s1>s2

The exact value returned in case of dissimilar strings is not defined. We only know that if s1<s2 then a negative value will be returned and if s1>s2 then a positive value will be returned.

For example:

```
/*String comparison… ..... */
#include<stdio.h>
#include<string.h>
void main()
{
        char str1[10],str2[10];
        printf("Enter two strings:");
        gets(str1);
        gets(str2);
        if(strcmp(str1,str2)==0)
{
        printf("String are same\n");
}
else
{
        printf("String are not same\n");
}
}
```

strcpy()

This function is used to copying one string to another string. The function strcpy(str1,str2) copies str2 to str1 including the NULL character. Here str2 is the source string and str1 is the destination string.

The old content of the destination string str1 are lost. The function returns a pointer to destination string str1.

Example:-

```
        #include<stdio.h>
        #include<string.h>
        void main()
{
        char str1[10],str2[10];
        printf("Enter a string:");
        scanf("%s",str2);
        strcpy(str1,str2);
        printf("First string:%s\t\tSecond string:%s\n",str1,str2);
        strcpy(str,"Delhi");
        strcpy(str2,"Bangalore");
printf("First string :%s\t\tSecond string:%s",str1,str2);
```

strcat()

This function is used to append a copy of a string at the end of the other string. If the first string is “”Purva” and second string is “Belmont” then after using this function the string becomes “PusvaBelmont”. The NULL character from str1 is moved and str2 is added at the end of str1. The 2nd string str2 remains unaffected. A pointer to the first string str1 is returned by the function.

Example:-

```
#include<stdio.h>
#include<string.h>
void main()
{
char str1[20],str[20];
printf(“Enter two strings:”);
gets(str1);
gets(str2);
strcat(str1,str2);
printf(“First string:%s\t second string:%s\n”,str1,str2);
strcat(str1,”-one”);
printf(“Now first string is %s\n”,str1);
}
```

Output

```
Enter two strings: data
Base
```

First string: database second string: database

` Now first string is: database-one

Lecture Note: 14

FUNCTION

A function is a self contained block of codes or sub programs with a set of statements that perform some specific task or coherent task when it is called.

It is something like to hiring a person to do some specific task like, every six months servicing a bike and hand over to it.

Any ‘C’ program contain at least one function i.e main().

There are basically two types of function those are

1. **Library function**
2. **User defined function**

The user defined functions defined by the user according to its requirement

System defined function can’t be modified, it can only read and can be used. These function are supplied with every C compiler

Source of these library function are pre complied and only object code get used by the user by linking to the code by linker

Here in system defined function description:

Function definition : predefined, precompiled, stored in the library

Function declaration : In header file with or function prototype.

Function call : By the programmer

User defined function

Syntax:-

Return type name of function (type 1 arg 1, type2 arg2, type3 arg3)

Return type function name argument list of the above syntax

So when user gets his own function three thing he has to know, these are.

Function declaration

Function definition

Function call

These three things are represented like

```
int function(int, int, int);       /*function declaration*/
main()       /* calling function*/
  {
function(arg1,arg2,arg3);
 }
 int function(type 1 arg 1,type2 arg2,type3, arg3)   /*function definition/*
{
 Local variable declaration;
Statement;
Return value;
 }
```

Function declaration:-

Function declaration is also known as function prototype. It inform the compiler about three thing, those are name of the function, number and type of argument received by the function and the type of value returned by the function.

While declaring the name of the argument is optional and the function prototype always terminated by the semicolon.

Function definition:-

Function definition consists of the whole description and code of the function.

It tells about what function is doing what are its inputs and what are its out put

It consists of two parts function header and function body

Syntax:-

```
return type function(type 1 arg1, type2 arg2, type3 arg3) /*function header*/
 {
   Local variable declaration;
   Statement 1;
   Statement 2;
   Return value
 }
```

The return type denotes the type of the value that function will return and it is optional and if it is omitted, it is assumed to be int by default. The body of the function is the compound statements or block which consists of local variable declaration statement and optional return statement.

The local variable declared inside a function is local to that function only. It can't be used anywhere in the program and its existence is only within this function.

The arguments of the function **definition** are known as **formal arguments**.

Function Call

When the function get called by the calling function then that is called, function call. The compiler execute these functions when the semicolon is followed by the function name.

Example:-

```
function(arg1,arg2,arg3);
```

The argument that are used inside the function call are called **actual argument**

Ex:-

```
int S=sum(a, b);                    //actual arguments
```

Actual argument

The arguments which are mentioned or used inside the function call is knows as actual argument and these are the original values and copy of these are actually sent to the called function

It can be written as constant, expression or any function call like

```
Function (x);

Function (20, 30);

Function (a*b, c*d);

Function(2,3,sum(a, b));
```

Formal Arguments

The arguments which are mentioned in function definition are called formal arguments or dummy arguments.

These arguments are used to just hold the copied of the values that are sent by the calling function through the function call.

These arguments are like other local variables which are created when the function call starts and destroyed when the function ends.

The basic difference between the formal argument and the actual argument are

1) The formal argument are declared inside the parenthesis where as the local variable declared at the beginning of the function block.

2). The **formal argument** are automatically initialized when the copy of actual arguments are passed while other local variable are assigned values through the statements.

Order number and type of actual arguments in the function call should be match with the order number and type of the formal arguments.

Return type

It is used to return value to the calling function. It can be used in two way as

```
return
Or      return(expression);
Ex:-    return (a);
        return (a*b);
        return (a*b+c);
```

Here the 1st return statement used to terminate the function without returning any value

Ex:- /*summation of two values*/

```
int sum (int a1, int a2);
main()
```

```
{
  int a,b;
   printf("enter two no");
   scanf("%d%d",&a,&b);
   int S=sum(a,b);
    printf("summation is = %d",s);
     }
    int sum(intx1,int y1)
     {
   int z=x1+y1;
    Return z;
    }
```

Advantage of function

By using function large and difficult program can be divided in to sub programs and solved. When we want to perform some task repeatedly or some code is to be used more than once at different place in the program, then function avoids this repeatition or rewritten over and over.

Due to reducing size, modular function it is easy to modify and test

Notes:-

C program is a collection of one or more function.

A function is get called when function is followed by the semicolon.

A function is defined when a function name followed by a pair of curly braces

Any function can be called by another function even main() can be called by other function.

```
main()
{
function1()
}
function1()
{
Statement;
function2;
}
function 2()
{


}
```

So every function in a program must be called directly or indirectly by the main() function. A function can be called any number of times.

A function can call itself again and again and this process is called **recursion**.

A function can be called from other function **but** a function can't be defined in another function

Lecture Note: 15

Category of Function based on argument and return type

i) Function with no argument & no return value

Function that have no argument and no return value is written as:-

```
void  function(void);
main()
{
void  function()
{
Statement;
}
```

Example:-

```
void me();
main()
{
me();
printf("in  main");
}
void   me()
{
printf("come on");
}
```

Output: come on

inn main

ii) Function with no argument but return value

Syntax:-

```
int   fun(void);
main()
{
    int r;
    r=fun();
  }
    int   fun()
   {
   reurn(exp);
    }
```

Example:-

```
int   sum();
main()
{
int   b=sum();
printf("entered   %d\n, b");
 }
int   sum()
  {
int  a,b,s;
```

```
s=a+b;
return s;
}
```

Here called function is independent and are initialized. The values aren't passed by the calling function .Here the calling function and called function are communicated partly with each other.

Lecture Note: 16

iii) function with argument but no return value

Here the function have argument so the calling function send data to the called function but called function dose n't return value.

Syntax:-

```
void   fun (int,int);
main()
{
int (a,b);
}
void   fun(int x, int y);
{
Statement;
}
```

Here the result obtained by the called function.

iv) function with argument and return value

Here the calling function has the argument to pass to the called function and the called function returned value to the calling function.

Syntax:-

```
fun(int,int);
 main()
{
   int r=fun(a,b);
}
     int  fun(intx,inty)
{
          return(exp);
}
```

Example:

```
main()
{
int fun(int);
int  a,num;
printf("enter value:\n");
scanf("%d",&a)
```

```
int num=fun(a);
  }
 int fun(int x)
 {
  ++x;
          return x;
  }
```

Call by value and call by reference

There are two way through which we can pass the arguments to the function such as **call by** value and **call by reference**.

1. Call by value

In the call by value copy of the actual argument is passed to the formal argument and the operation is done on formal argument.

When the function is called by 'call by value' method, it doesn't affect content of the actual argument.

Changes made to formal argument are local to block of called function so when the control back to calling function the changes made is vanish.

Example:-

```
  main()
{
 int x,y;
change(int,int);
```

```
    printf("enter two values:\n");
     scanf("%d%d",&x,&y);
     change(x ,y);
   printf("value of x=%d and y=%d\n",x ,y);
    }
   change(int a,int b);
  {
   int k;
   k=a;
   a=b;
   b=k;
  }
```

Output: enter two values: 12

23

Value of x=12 and y=23

2. Call by reference

Instead of passing the value of variable, address or reference is passed and the function operate on address of the variable rather than value.

Here formal argument is alter to the actual argument, it means formal arguments calls the actual arguments.

Example:-

```
    void main()
```

```
{
        int a,b;
        change(int *,int*);
        printf("enter two values:\n");
        scanf("%d%d",&a,&b);
        change(&a,&b);
        printf("after changing two value of a=%d and b=%d\n:"a,b);
}
        change(int *a, int *b)
        {
        int k;
          k=*a;
        *a=*b;
      *b= k;
printf("value in this function a=%d and b=%d\n",*a,*b);
}
```

Output: enter two values: 12

32

Value in this function a=32 and b=12

After changing two value of a=32 and b=12

So here instead of passing value of the variable, directly passing address of the variables. Formal argument directly access the value and swapping is possible even after calling a function.

Lecture Note: 17

Local, Global and Static variable

Local variable:-

variables that are defined with in a body of function or block. The local variables can be used only in that function or block in which they are declared. Same variables may be used in different functions such as

```
function()
{
        int a,b;
        function 1();
}
function2 ()
{
        int a=0;
        b=20;
}
```

Global variable:-

the variables that are defined outside of the function is called global variable. All functions in the program can access and modify global variables. Global variables are automatically initialized at the time of initialization.

Example:

```
#include<stdio.h>
void function(void);
void function1(void);
void function2(void);
int a, b=20;
void main()
{
printf("inside main a=%d,b=%d \n",a,b);
function();
function1();
function2();
}
 function()
{
Prinf("inside function a=%d,b=%d\n",a,b);
}
function 1()
{
```

```
        prinf("inside function a=%d,b=%d\n",a,b);
}
function 2()
{
        prinf("inside function a=%d,b=%d\n",a,);
}
```

Static variables: static variables are declared by writing the key word static.

-syntax:-

```
static data type variable name;
static int a;
```

-the static variables initialized only once and it retain between the function call. If its variable is not initialized, then it is automatically initialized to zero.

Example:

```
void fun1(void);
void fun2(void);
void main()
{
        fun1();
        fun2();
}
void fun1()
{
```

```
        int a=10, static int b=2;
        printf("a=%d, b=%d",a,b);
        a++;
        b++;
    }
```

Output:a= 10 b= 2

a=10 b= 3

Recursion

When function calls itself (inside function body) again and again then it is called as recursive function. In recursion calling function and called function are same. It is powerful technique of writing complicated algorithm in easiest way. According to recursion problem is defined in term of itself. Here statement with in body of the function calls the same function and same times it is called as circular definition. In other words recursion is the process of defining something in form of itself.

Syntax:

```
main ()
{
    rec(); /*function call*/
    rec();
    rec();
```

Ex:-

```
/*calculate factorial of a no.using recursion*/
int fact(int);
void main()
```

```
{
        int num;
        printf("enter a number");
        scanf("%d",&num);
        f=fact(num);
        printf("factorial is =%d\n"f);
}
fact (int num)
{
        If  (num==0||num==1)
return 1;
else
return(num*fact(num-1));
}
```

Lecture Note: 18

Monolithic Programming

The program which contains a single function for the large program is called monolithic program. In monolithic program not divided the program, it is huge long pieces of code that jump back and forth doing all the tasks like single thread of execution, the program requires. Problem arise in monolithic program is that, when the program **size** increases it leads inconvenience and difficult to maintain

such as testing, debugging etc. Many disadvantages of monolithic programming are:

1. Difficult to check error on large programs size.
2. Difficult to maintain because of huge size.
3. Code can be specific to a particular problem. i.e. it cannot be reused.

Many early languages (FORTRAN, COBOL, BASIC, C) required one huge workspace with labelled areas that may does specific tasks but are not isolated.

Modular Programming

The process of subdividing a computer program into separate sub-programs such as functions and subroutines is called Modular programming. **Modular programming sometimes also called as structured programming.** It enables multiple programmers to divide up the large program and debug pieces of program independently and tested.

. Then the linker will link all these modules to form the complete program. This principle dividing software up into parts, or modules, where a module can be changed, replaced, or removed, with minimal effect on the other software it works with. Segmenting the program into modules clearly defined functions, it can determine the source of program errors more easily. Breaking down program functions into modules, where each of which accomplishes one function and contains all the source code and variables needed to accomplish that function. Modular program is the solution to the problem of very large program that are difficult to debug, test and maintain. A program module may be rewritten while its inputs and outputs remain the same. The person making a change may only understand a small portion of the original program.

Object-oriented programming (OOP) is compatible with the modular programming concept to a large extent.

. , Less code has to be written that makes shorter.

- A single procedure can be developed for reuse, eliminating the need to retype the code many times.
- Programs can be designed more easily because a small team deals with only a small part of the entire code.
- Modular programming allows many programmers to collaborate on the same application.
- The code is stored across multiple files.
- Code is short, simple and easy to understand and modify, make simple to figure out how the program is operate and reduce likely hood of bugs.
- Errors can easily be identified, as they are localized to a subroutine or function or isolated to specific module.
- The same code can be reused in many applications.
- The scoping of variables and functions can easily be controlled.

Disadvantages
However it may takes longer to develop the program using this technique.

Storage Classes

Storage class in c language is a specifier which tells the compiler where and how to store variables, its initial value and scope of the variables in a program. Or attributes of variable is known as storage class or in compiler point of view a variable identify some physical location within a computer where its string of bits value can be stored is known as storage class.

The kind of location in the computer, where value can be stored is either in the memory or in the register. There are various storage class which determined, in which of the two location value would be stored.

Syntax of declaring storage classes is:-

storageclass datatype variable name;

There are four types of storage classes and all are keywords:-

1) Automatic (auto)

2) Register (register)

3) Static (static)

4) External (extern)

Examples:-

auto float x; or float x;

extern int x;

register char c;

static int y;

Compiler assume different storage class based on:-

1) Storage class:- tells us about storage place(where variable would be stored).

2) Intial value :-what would be the initial value of the variable.

If initial value not assigned, then what value taken by uninitialized variable.

3) Scope of the variable:-what would be the value of the variable of the program.

4) Life time :- It is the time between the creation and distribution of a variable or how long would variable exists.

1. Automatic storage class

The keyword used to declare automatic storage class is auto.

Its features:-

Storage-memory location

Default initial value:-unpredictable value or garbage value.

Scope:-local to the block or function in which variable is defined.

Life time:-Till the control remains within function or block in which it is defined. It terminates when function is released.

The variable without any storage class specifier is called automatic variable.

Example:-

```
main( )
{
auto int i;
printf("i=",i);
}
```

Lecture Note: 19

2. Register storage class

The keyword used to declare this storage class is register.

The features are:-

Storage:-CPU register.

Default initial value :-garbage value

Scope :-local to the function or block in which it is defined.

Life time :-till controls remains within function or blocks in which it is defined.

Register variable don't have memory address so we can't apply address operator on it. CPU register generally of 16 bits or 2 bytes. So we can apply storage classes only for integers, characters, pointer type.

Variable stored in register storage class always access faster than,which is always stored in the memory. But to store all variable in the CPU register is not possible because of limitation of the register pair.

And when variable is used at many places like loop counter, then it is better to declare it as register class.

Example:-

```
main( )
{
register int i;
for(i=1;i<=12;i++)
printf("%d",i);
}
```

3 Static storage class

The keyword used to declare static storage class is static.

Its feature are:-

Storage:-memory location

Default initial value:- zero

Scope :- local to the block or function in which it is defined.

Life time:- value of the variable persist or remain between different function call.

Example:-

```
main( )
```

```
{
reduce( );
reduce( );
reduce ( );
}
reduce( )
{
static int x=10;
printf("%d",x);
x++;
}
```

Output:-10,11,12

External storage classes

The keyword used for this class is extern.

Features are:-

Storage:- memory area

Default initial value:-zero

Scope :- global

Life time:-as long as program execution remains it retains.

Declaration does not create variables, only it refer that already been created at somewhere else. So, memory is not allocated at a time of declaration and the external variables are declared at outside of all the function.

Example:-

```
int i,j;
void main( )
{
printf( “i=%d”,i );
receive( );
receive ( );
reduce( );
reduce( );
}
receive( )
{
i=i+2;
printf(“on increase i=%d”,i);
}
reduce( )
{
i=i-1;
printf(“on reduce i=%d”,i);
}
```

Output:-i=0,2,4,3,2.

When there is large program i.e divided into several files, then external variable should be preferred. External variable extend the scope of variable.

Lecture Note: 20

POINTER

A pointer is a variable that store memory address or that contains address of another variable where addresses are the location number always contains whole number. So, pointer contain always the whole number. It is called pointer because it points to a particular location in memory by storing address of that location.

Syntax-

Data type *pointer name;

Here * before pointer indicate the compiler that variable declared as a pointer.

e.g.

int *p1; //pointer to integer type

float *p2; //pointer to float type

char *p3; //pointer to character type

When pointer declared, it contains garbage value i.e. it may point any value in the memory.

Two operators are used in the pointer i.e. **address operator(&)** and **indirection operator or dereference operator (*).**

Indirection operator gives the values stored at a particular address.

Address operator cannot be used in any constant or any expression.

Example:

```
    void main()
    {
      int i=105;
      int *p;
      p=&i;
t
printf("value of i=%d",*p);
printf("value of i=%d",*/&i);
printf("address of i=%d",&i);
printf("address of i=%d",p);
printf("address of p=%u",&p);
}
```

Pointer Expression

Pointer assignment

int i=10;

int *p=&i;//value assigning to the pointer

Here declaration tells the compiler that P will be used to store the address of integer value or in other word P is a pointer to an integer and *p reads the **value at the address contain in p.**

P++;

printf("value of p=%d");

We can assign value of 1 pointer variable to other when their base type and data type is same or both the pointer points to the same variable as in the array.

Int *p1,*p2;

P1=&a[1];

P2=&a[3];

We can assign constant 0 to a pointer of any type for that symbolic constant '**NULL**' is used such as

*p=NULL;

It means pointer doesn't point to any valid memory location.

Pointer Arithmetic

Pointer arithmetic is different from ordinary arithmetic and it is perform relative to the data type(base type of a pointer).

Example:-

If integer pointer contain address of 2000 on incrementing we get address of 2002 instead of 2001, because, size of the integer is of 2 bytes.

Note:-

When we move a pointer, somewhere else in memory by incrementing or decrement or adding or subtracting integer, it is not necessary that, pointer still pointer to a variable of same data, because, memory allocation to the variable are done by the compiler.

But in case of array it is possible, since there data are stored in a consecutive manner.

Ex:-

```
void main( )
{
static int a[ ]={20,30,105,82,97,72,66,102};
int *p,*p1;
P=&a[1];
P1=&a[6];
printf("%d",*p1-*p);
printf("%d",p1-p);
}
```

Arithmetic operation never perform on pointer are:

addition, multiplication and division of two pointer.

multiplication between the pointer by any number.

division of pointer by any number

-add of float or double value to the pointer.

Operation performed in pointer are:-

/* Addition of a number through pointer */

Example

```
int i=100;
int *p;
```

p=&i;

p=p+2;

p=p+3;

p=p+9;

ii /* Subtraction of a number from a pointer'*/

Ex:-

int i=22;

*p1=&a;

p1=p1-10;

p1=p1-2;

iii- Subtraction of one pointer to another is possible when pointer variable point to an element of same type such as an array.

Ex:-

in tar[ ]={2,3,4,5,6,7};

int *ptr1,*ptr1;

ptr1=&a[3]; //2000+4

ptr2=&a[6]; //2000+6

Lecture Note: 21

Precedence of dereference (*) Operator and increment operator and decrement operator

The precedence level of difference operator increment or decrement operator is same and their associatively from right to left.

Example :-

int x=25;

int *p=&x;

Let us calculate int y=*p++;

Equivalent to *(p++)

Since the operator associate from right to left, increment operator will applied to the pointer p.

i) int y=*p++; equivalent to *(p++)

p =p++ or p=p+1

ii) *++p;→*(++p)→p=p+1

y=*p

iii) int y=++*p

equivalent to ++(*p)

p=p+1 then *p

iv) y=(*p)++→equivalent to *p++

y=*p then

P=p+1 ;

Since it is postfix increment the value of p.

Pointer Comparison

Pointer variable can be compared when both variable, object of same data type and it is useful when both pointers variable points to element of same array.

Moreover pointer variable are compared with zero which is usually expressed as null, so several operators are used for comparison like the relational operator.

==,!=,<=,<,>,>=, can be used with pointer. Equal and not equal operators used to compare two pointer should finding whether they contain same address or not and they will equal only if are null or contains address of same variable.

Ex:-

```
  void main()
{
static int arr[]={20,25,15,27,105,96}
int *x,*y;
x=&a[5];
y=&(a+5);
if(x==y)
printf(“same”);
else
printf(“not”);



}
```

Lecture Note: 22

Pointer to pointer

Addition of pointer variable stored in some other variable is called pointer to pointer variable.

Or

Pointer within another pointer is called pointer to pointer.

Syntax:-

Data type **p;

int x=22;

int *p=&x;

int **p1=&p;

printf("value of x=%d",x);

printf("value of x=%d",*p);

printf("value of x=%d",*&x);

printf("value of x=%d",**p1);

printf("value of p=%u",&p);

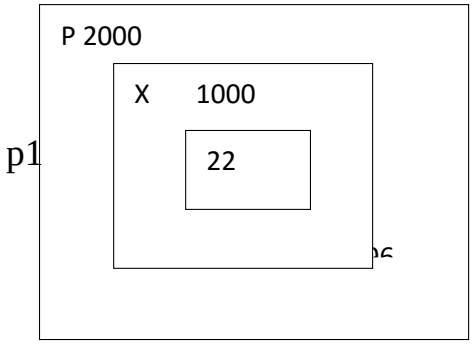
printf("address of p=%u",p1);

printf("address of x=%u",p);

printf("address of p1=%u",&p1);

printf("value of p=%u",p);

printf("value of p=%u",&x);



3000

Pointer vs array

Example :-

```
   void main()
{
static char arr[]=”Rama”;
char*p=”Rama”;
printf(“%s%s”, arr, p);
```

In the above example, at the first time printf( ), print the same value array and pointer.

Here array arr, as **pointer to character** and **p act as a pointer to array of character .** When we are trying to increase the value of arr it would give the error because its known to compiler about an array and its base address which is always printed to base address is known as constant pointer and the base address of array which is not allowed by the compiler.

printf(“size of (p)”,size of (ar));

size of (p) 2/4 bytes

size of(ar) 5 byes

Sructure

It is the collection of dissimilar data types or heterogenous data types grouped together. It means the data types may or may not be of same type.

Structure declaration-

struct tagname

{

Data type member1;

Data type member2;

Data type member3;

……

……

Data type member n;

};

OR

struct

{

Data type member1;

Data type member2;

```
Data type member3;
……
……
Data type member n;
};
OR
struct tagname
{
struct element 1;
struct element 2;
struct element 3;
……
……
struct element n;
};
```

Structure variable declaration;

```
struct student
{
        int age;
char name[20];
char branch[20];
```

}; struct student s;

Initialization of structure variable-

Like primary variables structure variables can also be initialized when they are declared. Structure templates can be defined locally or globally. If it is local it can be used within that function. If it is global it can be used by all other functions of the program.

We cant initialize structure members while defining the structure

```
struct student
{
        int age=20;
char name[20]=”sona”;
}s1;
```

The above is **invalid.**

A structure can be initialized as

```
struct student
{
        int age,roll;
char name[20];
} struct student s1={16,101,”sona”};
  struct student s2={17,102,”rupa”};
```

If initialiser is less than no.of structure variable, automatically rest values are taken as zero.

Accessing structure elements-

Dot operator is used to access the structure elements. Its associativety is from left to right.

structure variable ;

s1.name[];

s1.roll;

s1.age;

Elements of structure are stored in contiguous memory locations. Value of structure variable can be assigned to another structure variable of same type using assignment operator.

Example:

```
#include<stdio.h>

#include<conio.h>

void main()

{

int roll, age;

char branch;

} s1,s2;

printf("\n enter roll, age, branch=");

scanf("%d %d %c", &s1.roll, &s1.age, &s1.branch);

s2.roll=s1.roll;

printf(" students details=\n");

printf("%d %d %c", s1.roll, s1.age, s1.branch);

printf("%d", s2.roll);
```

}

Unary, relational, arithmetic, bitwise operators are not allowed within structure variables.

Lecture Note:24

Size of structure-

Size of structure can be found out using sizeof() operator with structure variable name or tag name with keyword.

sizeof(struct student); or

sizeof(s1);

sizeof(s2);

Size of structure is different in different machines. So size of whole structure may not be equal to sum of size of its members.

Array of structures

When database of any element is used in huge amount, we prefer Array of structures.

Example: suppose we want to maintain data base of 200 students, Array of structures is used.

#include<stdio.h>

#include<string.h>

struct student

{

```
char name[30];
char branch[25];
int roll;
};
void main()
{
struct student s[200];
int i;
s[i].roll=i+1;
printf("\nEnter information of students:");
for(i=0;i<200;i++)
{
printf("\nEnter the roll no:%d\n",s[i].roll);
printf("\nEnter the name:");
scanf("%s",s[i].name);
printf("\nEnter the branch:");
scanf("%s",s[i].branch);
printf("\n");
}
printf("\nDisplaying information of students:\n\n");
for(i=0;i<200;i++)
{
printf("\n\nInformation for roll no%d:\n",i+1);
```

```
printf("\nName:");
puts(s[i].name);
printf("\nBranch:");
puts(s[i].branch);
}
}
```

In Array of structures each element of array is of structure type as in above example.

Array within structures

```
struct student
{
char name[30];
int roll,age,marks[5];
}; struct student s[200];
```

We can also initialize using same syntax as in array.

Nested structure

When a structure is within another structure, it is called Nested structure. A structure variable can be a member of another structure and it is represented as

```
struct student
```

```
{
element 1;
element 2;
……
……
struct student1
{
member 1;
member 2;
}variable 1;
…….
…….
element n;
}variable 2;
```

It is possible to define structure outside & declare its variable inside other structure.

```
struct date
{
int date,month;
};
struct student
{
```

```
 char nm[20];
int roll;
struct date d;
}; struct student s1;
   struct student s2,s3;
```

Nested structure may also be initialized at the time of declaration like in above example.

```
struct student s={“name”,200, {date, month}};
                  {“ram”,201, {12,11}};
```

Nesting of structure within itself is not valid. Nesting of structure can be extended to any level.

```
struct time
{
int hr,min;
};
struct day
{
int date,month;
struct time t1;
};
struct student
```

```
{
char nm[20];
struct day d;
}stud1, stud2, stud3;
```

Lecture Note: 25

Passing structure elements to function

We can pass each element of the structure through function but passing individual element is difficult when number of structure element increases. To overcome this, we use to pass the whole structure through function instead of passing individual element.

```
#include<stdio.h>
#include<string.h>
void main()
{
struct student
{
char name[30];
char branch[25];
int roll;
}struct student s;
printf("\n enter name=");
```

```
gets(s.name);
printf("\nEnter roll:");
scanf("%d",&s.roll);
printf("\nEnter branch:");
gets(s.branch);
display(name,roll,branch);
}
display(char name, int roll, char branch)
{
printf("\n name=%s,\n roll=%d, \n branch=%s", s.name, s.roll. s.branch);
}
```

Passing entire structure to function

```
#include<stdio.h>
#include<string.h>
struct student
{
char name[30];
int age,roll;
};
display(struct student);                    //passing entire structure
void main()
```

```
{
   struct student s1={”sona”,16,101 };
   struct student s2={”rupa”,17,102 };
display(s1);
display(s2);
}
display(struct student s)
{
printf(“\n name=%s, \n age=%d ,\n roll=%d”, s.name, s.age, s.roll);
}
```

Output: name=sona

roll=16

Lecture Note: 26

UNION

Union is derived data type contains collection of different data type or dissimilar elements. All definition declaration of union variable and accessing member is similar to structure, but instead of keyword struct the keyword union is used, the main difference between union and structure is

Each member of structure occupy the memory location, but in the unions members share memory. Union is used for saving memory and concept is useful when it is not necessary to use all members of union at a time.

Where union offers a memory treated as variable of one type on one occasion where (struct), it read number of different variables stored at different place of memory.

Syntax of union:

union student

{

datatype member1;

datatype member2;

};

Like structure variable, union variable can be declared with definition or separately such as

union union name

{

Datatype member1;

}var1;

Example:- union student s;

Union members can also be accessed by the dot operator with union variable and if we have pointer to union then member can be accessed by using (arrow) operator as with structure.

Example:- struct student

struct student

{

int i;

char ch[10];

};struct student s;

Here datatype/member structure occupy 12 byte of location is memory, where as in the union side it occupy only 10 byte.

Lecture Note:27

Nested of Union

When one union is inside the another union it is called nested of union.

Example:-

union a

{

int i;

int age;

};

union b

```
{
char name[10];
union a aa;
}; union b bb;
```

There can also be union inside structure or structure in union.

Example:-

```
 void main()
   {
  struct a
 {
int i;
char ch[20];
};
struct  b
{
int i;
char d[10];
};
union z
{
struct a a1;
struct b b1;
```

```
}; union z z1;

z1.b1.j=20;

z1.a1.i=10;

z1.a1.ch[10]= " i";

z1.b1.d[0]="j ";

printf(" ");
```

Dynamic memory Allocation

The process of allocating memory at the time of execution or at the runtime, is called dynamic memory location.

Two types of problem may occur in static memory allocation.

If number of values to be stored is less than the size of memory, there would be wastage of memory.

If we would want to store more values by increase in size during the execution on assigned size then it fails.

Allocation and release of memory space can be done with the help of some library function called dynamic memory allocation function. These library function are called as **dynamic memory allocation function.** These library function prototype are found in the header file, "alloc.h" where it has defined.

Function take memory from memory area is called heap and release when not required.

Pointer has important role in the dynamic memory allocation to allocate memory.

malloc():

This function use to allocate memory during run time, its declaration is void*malloc(size);

malloc ()

returns the pointer to the 1st byte and allocate memory, and its return type is void, which can be type cast such as:

int *p=(datatype*)malloc(size)

If memory location is successful, it returns the address of the memory chunk that was allocated and it returns null on unsuccessful and from the above declaration a pointer of type**(datatype)** and size in byte.

And **datatype** pointer used to typecast the pointer returned by malloc and this typecasting is necessary since, malloc() by default returns a pointer to void.

Example int*p=(int*)malloc(10);

So, from the above pointer p, allocated IO contigious memory space address of 1st byte and is stored in the variable.

We can also use, the size of operator to specify the the size, such as *p=(int*)malloc(5*size of int) Here, 5 is the no. of data.

Moreover , it returns null, if no sufficient memory available , we should always check the malloc return such as, **if(p==null)**

printf(“not sufficient memory”);

Example:

/*calculate the average of mark*/

void main()

{

int n , avg,i,*p,sum=0;

```
printf("enter the no. of marks ”);
scanf(“%d”,&n);
p=(int *)malloc(n*size(int));
if(p==null)
printf(“not sufficient”);
exit();
}
for(i=0;i<n;i++)
scanf(“%d”,(p+i));
for(i=0;i<n;i++)
Printf(“%d”,*(p+i));
sum=sum+*p;
avg=sum/n;
printf(“avg=%d”,avg);
```

Lecture Note: 28

calloc()

Similar to malloc only difference is that calloc function use to allocate multiple block of memory .

two arguments are there

1st argument specify number of blocks

2nd argument specify size of each block.

Example:-

int *p= (int*) calloc(5, 2);

int*p=(int *)calloc(5, size of (int));

Another difference between malloc and calloc is by default memory allocated by malloc contains garbage value, where as memory allocated by calloc is initialised by zero(but this initialisation) is not reliable.

realloc()

The function realloc use to change the size of the memory block and it alter the size of the memory block without loosing the old data, it is called reallocation of memory.

It takes two argument such as;

int *ptr=(int *)malloc(size);

int*p=(int *)realloc(ptr, new size);

The new size allocated may be larger or smaller.

If new size is larger than the old size, then old data is not lost and newly allocated bytes are uninitialized. If old address is not sufficient then starting address contained in pointer may be changed and this reallocation function moves content of old block into the new block and data on the old block is not lost.

Example:

#include<stdio.h>

#include<alloc.h>

void main()

int i,*p;

```
p=(int*)malloc(5*size of (int));
if(p==null)
{
printf(“space not available”);
exit();
printf(“enter 5 integer”);
for(i=0;i<5;i++)
{
scanf(“%d”,(p+i));
int*ptr=(int*)realloc(9*size of (int) );
if(ptr==null)
{
printf(“not available”);
exit();
}
printf(“enter 4 more integer”);
for(i=5;i<9;i++)
scanf(“%d”,(p+i));
for(i=0;i<9;i++)
                    printf(“%d”,*(p+i));
}
```

free()

Function free() is used to release space allocated dynamically, the memory released by free() is made available to heap again. It can be used for further purpose.

Syntax for free declaration .

void(*ptr)

Or

free(p)

When program is terminated, memory released automatically by the operating system. Even we don't free the memory, it doesn't give error, thus lead to memory leak.

We can't free the memory, those didn't allocated.

Lecture Note: 29

Dynamic array

Array is the example where memory is organized in contiguous way, in the dynamic memory allocation function used such as malloc(), calloc(), realloc() always made up of contiguous way and as usual we can access the element in two ways as:

Subscript notation

Pointer notation

Example:

```
#include<stdio.h>
#include<alloc.h>
void main()
{
printf(“enter the no.of values”);
scanf(“%d”,&n);
p=(int*)malloc(n*size of int);
If(p==null)
printf(“not available memory”);
exit();
}
for(i=0;i<n;i++)
{
printf(“enter an integer”);
scanf(“%d”,&p[i]);
for(i=0;i<n;i++)
{
printf(“%d”,p[i]);
}
}
```

File handling

File: the file is a permanent storage medium in which we can store the data permanently.

Types of file can be handled

we can handle three type of file as

(1) sequential file

(2) random access file

(3) binary file

File Operation

opening a file:

Before performing any type of operation, a file must be opened and for this fopen() function is used.

syntax:

file-pointer=fopen("FILE NAME ","Mode of open");

example:

```
FILE *fp=fopen("ar.c","r");
```

If fopen() unable to open a file than it will return NULL to the file pointer.

File-pointer: The file pointer is a pointer variable which can be store the address of a special file that means it is based upon the file pointer a file gets opened.

Declaration of a file pointer:-

FILE* var;

Modes of open

The file can be open in three different ways as

Read mode ' r'/rt

Write mode 'w'/wt

Appened Mode 'a'/at

Reading a character from a file

getc() is used to read a character into a file

Syntax:

character_variable=getc(file_ptr);

Writing acharacter into a file

putc() is used to write a character into a file

puts(character-var,file-ptr);

ClOSING A FILE

fclose() function close a file.

fclose(file-ptr);

fcloseall () is used to close all the opened file at a time

File Operation

The following file operation carried out the file

(1)creation of a new file

(3)writing a file

(4)closing a file

Before performing any type of operation we must have to open the file.c, language communicate with file using A new type called **file pointer**.

Operation with fopen()

File pointer=fopen("FILE NAME","mode of open");

If **fopen()** unable to open a file then it will return **NULL** to the file-pointer.

Lecture Note: 30

Reading and writing a characters from/to a file

fgetc() is used for reading a character from the file

Syntax:

character variable= fgetc(file pointer);

fputc() is used to writing a character to a file

Syntax:

fputc(character,file_pointer);

```
/*Program to copy a file to another*/
#include<stdio.h>
void main()
{
FILE *fs,*fd;
char ch;
If(fs=fopen(“scr.txt”,”r”)==0)
{
printf(“sorry….The source file cannot be opened”);
return;
}
If(fd=fopen(“dest.txt”,”w”)==0)
{
printf(“Sorry…..The destination file cannot be opened”);
fclose(fs);
return;
}
while(ch=fgets(fs)!=EOF)
fputc(ch,fd);
fcloseall();
}
```

Reading and writing a string from/to a file

getw() is used for reading a string from the file

Syntax:

gets(file pointer);

putw() is used to writing a character to a file

Syntax:

fputs(integer,file_pointer);

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
FILE *fp;
int word;
/*place the word in a file*/
fp=fopen("dgt.txt","wb");
If(fp==NULL)
{
printf("Error opening file");
exit(1);
}
word=94;
putw(word,fp);
If(ferror(fp))
```

```
printf("Error writing to file\n");
else
printf("Successful write\n");
fclose(fp);
/*reopen the file*/
fp=fopen("dgt.txt","rb");
If(fp==NULL)
{
printf("Error opening file");
exit(1);
}

/*extract the word*/
word=getw(fp);
If(ferror(fp))
printf("Error reading file\n");
else
printf("Successful read:word=%d\n",word);
/*clean up*/
fclose(fp);
}
```

Lecture Note: 31

Reading and writing a string from/to a file

fgets() is used for reading a string from the file

Syntax:

fgets(string, length, file pointer);

fputs() is used to writing a character to a file

Syntax:

fputs(string,file_pointer);

```
#include<string.h>
#include<stdio.h>
void main(void)
{
FILE*stream;
char string[]=”This is a test”;
char msg[20];
/*open a file for update*/
stream=fopen(“DUMMY.FIL”,”w+”);

/*write a string into the file*/
fwrite(string,strlen(string),1,stream);
/*seek to the start of the file*/
fseek(stream,0,SEEK_SET);
```

```
/*read a string from the file*/
fgets(msg,strlen(string)+1,stream);
/*display the string*/
printf("%s",msg);
fclose(stream);
}
```