UNIT I BASICS OF C PROGRAMMING

Introduction to programming paradigms - Structure of C program - C programming: Data Types – Storage classes - Constants – Enumeration Constants - Keywords – Operators: Precedence and Associativity - Expressions - Input/Output statements, Assignment statements – Decision making statements - Switch statement - Looping statements – Pre-processor directives - Compilation process

Introduction to programming paradigms

The programming language „C" was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. Although C was initially developed for writing system software, today it has become such a popular language that a variety of software programs are written using this language. The greatest advantage of using C for programming is that it can be easily used on different types of computers. Many other programming languages such as C++ and Java are also based on C which means that you will be able to learn them easily in the future. Today, C is widely used with the UNIX operating system.

Programming Languages

The purpose of using computers is to solve problems, which are difficult to solve manually. So the computer user must concentrate on the development of good algorithms for solving problems. After preparing the algorithms, the programmer or user has to concentrate the programming language which is understandable by the computer.

Computer programming languages are developed with the primary objectives of facilitating a large number of people to use computer without the need to know the details of internal structure of the computer.

Structure of C program

A C program contains one or more functions, where a function is defined as a group of statements that perform a well-defined task. Figure 1.1 shows the structure of a C program. The statements in a function are written in a logical sequence to perform a specific task. The main()

function is the most important function and is a part of every C program. Rather, the execution of a C program begins with this function.

From the structure given in Fig. 1.1, we can conclude that a C program can have any number of functions depending on the tasks that have to be performed, and each function can have any number of statements arranged according to specific meaningful sequence. Note that programmers can choose any name for functions. It is not mandatory to write Function1, Function2, etc., with an exception that every program must contain one function that has its name as main().

```
main()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
Function1()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
Function2()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
.................
.................
FunctionN()
{
        Statement 1;
        Statement 2;
        ............
        ............
        Statement N;
}
```


Fig 1.1 Structure of C program

C Programming : Data Types

Data type determines the set of values that a data item can take and the operations that can beperformed on the item. C language provides four basic data types. Table 1.2 lists the data types, their size, range, and usage for a C programmer.

Data Type	Size in Bytes	Range	Use
char	1	-128 to 127	To store characters
int	2	-32768 to 32767	To store integer numbers
float	4	3.4E-38 to 3.4E+38	To store floating point numbers
double	8	1.7E-308 to 1.7E+308	To store big floating point numbers

Basic Data Types in C

Data Type	Size in Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Basic data types and their variants

Storage classes

Storage classes of a variable determine:

1. Storage area of the variable
2. Initial value of variable if not initialized
3. Lifetime of the variable

4. Linkage of a function or variable

C language provides 4 storage class specifiers:

1. auto
2. register
3. static
4. extern

Syntax:

storageclassspecifier datatype variable name;

1. **The auto storage class**

- Automatic variables are also called as auto variables, which are defined inside a function.
- Auto variables are stored in main memory.
- Variables has automatic (local) lifetime.
- Auto variables are not implicitly initialized.
- Auto variables have no linkage.
- It is a default storage class if the variable is declared inside the block.

Example:

```
#include<stdio.h>
void main()
{
auto int a=10;
printf("a=%d",a);

{
int b=20;
printf("b=%d",b);
}
printf("Here b is not visible\n");
printf("a=%d",a);
}
```

Output

a=10

b=20

Here b is not visible

a=10

2. The register storage class

- Register variables can be accessed faster than others
- Variables are stored in CPU.
- Variables have automatic (local) lifetime.
- Register variables are not implicitly initialized.
- Register variables have no linkage.
- Register variables are used for loop counter to improve the performance of a program.

Example:

```
# include<stdio.h>
void main()
{
register int a=200;
printf("a=%d",a);
}
```


Output:

a=200

3. The static storage class

- Static variables have static (global) lifetime.
- Static variables are stored in the main memory.
- Static variables can be declared in local scope as well as in the global scope.
- Static variables are implicitly initialized.
- The value of static variables persists between the function calls. Last change made in the value of static variable remains throughout the program execution.

- Static variables will have internal linkage.

Ex:

```
# include<stdio.h>
void fun(int);
void main()
{
int i=0;
for(i=0;i<5;i++)
{
fun(i);
}
}
void fun(int i)
{
static int a=10;
a++;
printf("%d",a);
}
```


Output:

11 12 13 14 15

4. The extern storage class

- Variables that are available to all functions are called external or global variables.
- External variables are stored in main memory.
- External variables have static (global) lifetime.
- External variables have external linkage.
- External variables are implicitly initialized to 0.

Example:

```
extern int v=10;
void call1()
```

```
void main()
{
call1();
printf("In main v=%d",v);
}
void call1()
{
printf("In call1() v=%d",v);
}
```

Output:

In main v=10

In call1( ) v=10

Since v is a external variable it is visible and accessed in all functions.

The typedef storage class

typedef is used for creating an alias or synonym for a known type.

Syntax:

```
typedef knowntype synonym name;
```

Eg:

```
typedef char* cp;
cp c;    // is same as char * c;
```

cp and (char *) can be used interchangeably.

Summary of storage classes

S.No	Storage class	Storage	Initial value	Lifetime	Linkage

1	auto	Memory	Garbage	Automatic	NO
2	register	CPU register	Garbage	Automatic	NO
3	static	Memory	0	Static	Internal
4	extern	Memory	0	Static	External

Constants

A constant is an entity whose value can‟t be changed during the execution of a program.

Constants are classified as:

1. Literal constants
2. Qualified constants
3. Symbolic constants

Literal constants

Literal constant or just literal denotes a fixed value, which may be an integer, floating point number, character or a string.

Literal constants are of the following types.

1. Integer Literal constant
2. Floating point Literal constant
3. Character Literal constant
4. String Literal constant

Integer Literal constant

Integer Literal constants are integer values like -1, 2, 8 etc. The rules for writing integer literal constant are:

- An Integer Literal constant must have at least one digit
- It should not have any decimal point
- It can be either positive or negative.
- No special characters and blank spaces are allowed.
- A number without a sign is assumed as positive.
- Octal constants start with 0.

- Hexadecimal constant start with 0x or 0X

Floating point Literal constant

Floating point Literal constants are values like -23.1, 12.8, 4e8 etc.. It can be written in a fractional form or in an exponential form.

The rules for writing Floating point Literal constants in a fractional form:

- A fractional floating point Literal constant must have at least one digit.
- It should have a decimal point.
- It can be either positive or negative.
- No special characters and blank spaces are allowed.

The rules for writing Floating point Literal constants in an exponential form:

- A Floating point Literal constants in an exponential form has two parts: the mantissa part and the exponent part. Both are separated by e or E.
- The mantissa part can be either positive or negative. The default sign is positive.
- The mantissa part should have at least one digit.
- The mantissa part can have a decimal point.
- The exponent part should have at least one digit
- The exponent part cannot have a decimal point.
- No special characters and blank spaces are allowed.

Character Literal constant

A Character Literal constant can have one or at most two characters enclosed within single quotes. E.g, „A‟ , „a‟ , „ n „.

It is classified into:

- Printable character literal constant
- Non-Printable character literal constant.

Printable character literal constant

All characters except quotation mark, backslash and new line characters enclosed within single quotes form a Printable character literal constant. Ex: „A‟ , „#‟

Non-Printable character literal constant.

Non-Printable character literal constants are represented with the help of escape sequences. An escape sequence consists of a backward slash (i.e. \) followed by a character and both enclosed within single quotes.

Constant	Meaning
'\a'	audible alert
'\b'	backspace
'\f'	form feed
'\n'	new line
'\0'	null
'\v'	vertical tab
'\t'	horizontal tab
'\r'	carriage return

String Literal constant

A String Literal constant consist of a sequence of characters enclosed within double quotes. Each string literal constant is implicitly terminated by a null character.

E.g. "ABC"

Qualified constants:

Qualified constants are created by using const qualifier.

E.g. const char a = „A"

The usage of const qualifier places a lock on the variable after placing the value in it. So we can"t change the value of the variable a

Symbolic constants

Symbolic constants are created with the help of the define preprocessor directive. For ex #define PI= 3.14 defines PI as a symbolic constant with the value 3.14. Each symbolic constant is replaced by its actual value during the pre-processing stage.

Keywords

Keyword is a reserved word that has a particular meaning in the programming language. The meaning of a keyword is predefined. It can"t be used as an identifier.

Keywords

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

Operators: Precedence and Associativity

Operands

An operand specifies an entity on which an operation is to be performed. It can be a variable name, a constant, a function call.

E.g: a=2+3 Here a, 2 & 3 are operands

Operator

An operator is a symbol that is used to perform specific mathematical or logical manipulations.

For e.g, a=2+3 Here = & + are the operators

Precedence of operators

- The precedence rule is used to determine the order of application of operators in evaluating sub expressions.
- Each operator in C has a precedence associated with it.
- The operator with the highest precedence is operated first.

Associativity of operators

The associativity rule is applied when two or more operators are having same precedence in the sub expression.

An operator can be left-to-right associative or right-to-left associative.

Category	Operators	Associativity	Precedence
Unary	+, -, !, ~, ++, - -, &, sizeof	Right to left	Level-1
Multiplicative	*, /, %	Left to right	Level-2

Additive	+, -	Left to right	Level-3
Shift	<< , >>	Left to right	Level-4
Relational	<, <=, >, >=	Left to right	Level-5

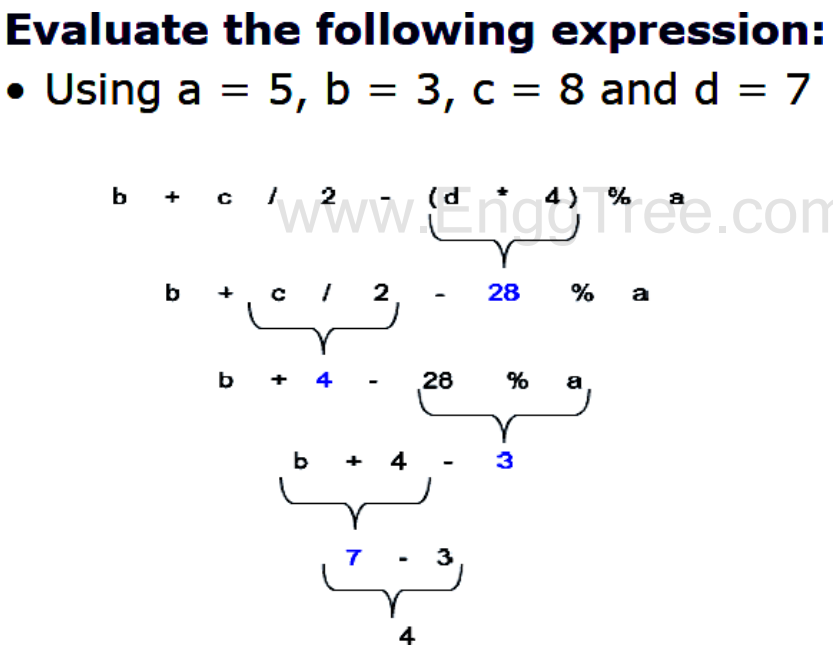
Rules for evaluation of expression

• First parenthesized sub expressions are evaluated first.

• If parentheses are nested, the evaluation begins with the innermost sub expression.

• The precedence rule is applied to determine the order of application of operators in evaluating sub expressions.

• The associability rule is applied when two or more operators are having same precedence in the



sub expression.

Expressions

An expression is a sequence of operators and operands that specifies computation of a value.

For e.g, a=2+3 is an expression with three operands a,2,3 and 2 operators = & +

Simple Expressions & Compound Expressions

An expression that has only one operator is known as a simple expression.

E.g: a+2

An expression that involves more than one operator is called a compound expression.

E.g: b=2+3*5

Classification of Operators

The operators in C are classified on the basis of

1) The number of operands on which an operator operates
2) The role of an operator

Classification based on Number of Operands

Types:

1. **Unary Operator**: A unary operator operates on only one operand. Some of the unary operators are,

Operator	Meaning
-	Minus
++	Increment
--	Decrement
&	Address- of operator
sizeof	sizeof operator

2. **Binary Operator**: A binary operator operates on two operands. Some of the binary operators are,

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication

/	Division
%	Modular Division
&&	Logical AND

3. **Ternary Operator**

A ternary operator operates on 3 operands. Conditional operator (i.e. **?:**) is the ternary operator.

Classification based on role of operands

Based upon their role, operators are classified as,

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Miscellaneous Operators

1. **Arithmetic Operators**

They are used to perform arithmetic operations like addition, subtraction, multiplication, division etc.

A=10 & B=20

Operator	**Description**	**Example**
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	The division operator is used to find the quotient.	B / A will give 2
%	Modulus operator is used to find the remainder	B%A will give 0
+,-	Unary plus & minus is used to indicate the algebraic sign of a value.	+A, -A

Increment operator

The operator ++ adds one to its operand.

- ++a or a++ is equivalent to a=a+1
- Prefix increment (++a) operator will increment the variable BEFORE the expression is evaluated.
- Postfix increment operator (a++) will increment AFTER the expression evaluation.
- E.g.
  1. c=++a. Assume a=2 so c=3 because the value of a is incremented and then it is assigned to c.
  2. d=b++ Assume b=2 so d=2 because the value of b is assigned to d before it is incremented.

Decrement operator

The operator – subtracts one from its operand.

- --a or a-- is equivalent to a=a+1
- Prefix decrement (--a) operator will decrement the variable BEFORE the expression is evaluated.
- Postfix decrement operator (a--) will decrement AFTER the expression evaluation.
- E.g.
  1. c=--a. Assume a=2 so c=1 because the value of a is decremented and then it is assigned to c.
  2. d=b-- Assume b=2 so d=2 because the value of b is assigned to d before it is decremented.

2. Relational Operators

- Relational operators are used to compare two operands. There are 6 relational operators in C, they are

Operator	Meaning of Operator	Example

==	Equal to	5==3 returns false (0)
>	Greater than	5>3 returns true (1)
<	Less than	5<3 returns false (0)
!=	Not equal to	5!=3 returns true(1)
>=	Greater than or equal to	5>=3 returns true (1)
<=	Less than or equal to	5<=3 return false (0)

- If the relation is true, it returns value 1 and if the relation is false, it returns value 0.
- An expression that involves a relational operator is called as a condition. For e.g a<b is a condition.

3. Logical Operators

- Logical operators are used to logically relate the sub-expressions. There are 3 logical operators in C, they are
- If the relation is true, it returns value 1 and if the relation is false, it returns value 0.

Operator	Meaning of Operator	Example	**Description**
&&	Logial AND	If c=5 and d=2 then,((c==5) && (d>5)) returns false.	It returns true when both conditions are true
||	Logical OR	If c=5 and d=2 then, ((c==5) || (d>5)) returns true.	It returns true when at-least one of the condition is true
!	Logical NOT	If c=5 then, !(c==5) returns false.	It reverses the state of the operand

Truth tables of logical operations

condition 1	**condition 2**	**NOT X**	**X AND Y**	**X OR Y**

(e.g., X)	(e.g., Y)	(~ X)	(X && Y)	(X || Y)
False	false	true	false	false
False	true	true	false	true
true	false	false	false	true
true	true	false	true	true

4. Bitwise Operators

C language provides 6 operators for bit manipulation. Bitwise operator operates on the individual bits of the operands. They are used for bit manipulation.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Truth tables

p	Q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

E.g.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

00001100

& 00011001

_____

00001000 = 8 (In decimal)

Bitwise OR Operation of 12 and 25

00001100

| 00011001

_____

00011101 = 29 (In decimal)

3 << 2 (Shift Left)

0011

1100=12

3 >> 1 (Shift Right)

0011

0001=1

5. Assignment Operators

To assign a value to the variable assignment operator is used.

Operators		**Example**	**Explanation**
Simple assignment operator	=	sum=10	10 is assigned to variable sum
Compound assignment operators Or Shorthand assignment operators	+=	sum+=10	This is same as sum=sum+10
	-=	sum-=10	sum = sum-10
	*=	sum*=10	sum = sum*10
	/+	sum/=10	sum = sum/10
	%=	sum%=10	sum = sum%10
	&=	sum&=10	sum = sum&10
	^=	sum^=10	sum = sum^10

E.g.

var=5 //5 is assigned to var

a=c; //value of c is assigned to a

6. Miscellaneous Operators

Other operators available in C are

a. Function Call Operator(i.e. ( ))
b. Array subscript Operator(i.e [ ])
c. Member Select Operator
    i. Direct member access operator (i.e. **.** dot operator)
    ii. Indirect member access operator (i.e. -> arrow operator)
d. Conditional operator (?:)
e. Comma operator (,)
f. sizeof operator
g. Address-of operator (&)

a) Comma operator

- It is used to join multiple expressions together.
- E.g.

```
int i , j;
i=(j=10,j+20);
```

In the above declaration, right hand side consists of two expressions separated by comma. The first expression is j=10 and second is j+20. These expressions are evaluated from left to right. ie first the value 10 is assigned to j and then expression j+20 is evaluated so the final value of i will be 30.

b) sizeof Operator

- The sizeof operator returns the size of its operand in bytes.
- Example : size of (char) will give us 1.
- sizeof(a), where a is integer, will return 2.

c) Conditional Operator

- It is the only ternary operator available in C.
- Conditional operator takes three operands and consists of two symbols ? and : .
- Conditional operators are used for decision making in C.

Syntax :

(Condition? true_value: false_value);

For example:

c=(c>0)?10:-10;

If *c* is greater than 0, value of *c* will be 10 but, if *c* is less than 0, value of *c* will be -10.

d) Address-of Operator

It is used to find the address of a data object.

Syntax

&operand

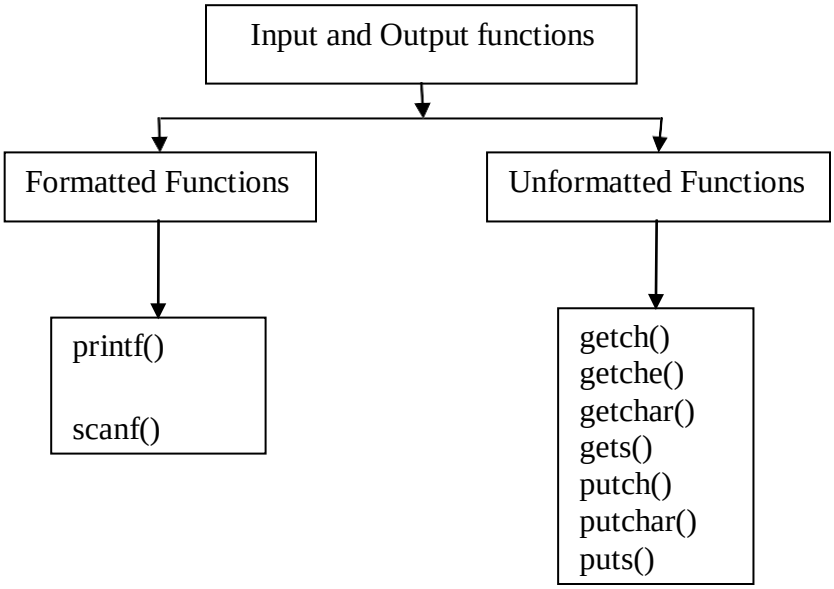
E.g.

&a will give address of a.

Input/Output statements

The I/O functions are classified into two types:

- Formatted Functions
- Unformatted functions



Unformatted Functions:

They are used when I/P & O/P is not required in a specific format.

C has 3 types I/O functions.

a) Character I/O
b) String I/O
c) File I/O

a) Character I/O:

1. ***getchar()*** This function reads a single character data from the standard input. (E.g. Keyboard)

Syntax :

```
variable_name=getchar();
```

eg:

```
char c;
c=getchar();
```

2. ***putchar()*** This function prints one character on the screen at a time.

Syntax :

```
putchar(variable name);
```

eg

```
char c=„C‟;
putchar(c);
```

Example Program

```
#include <stdio.h>
main()
{
int i;
char ch;
ch = getchar();
putchar(ch);
}
```

Output:

A

A

3. ***getch() and getche()*** : getch() accepts only a single character from keyboard. The character entered through getch() is not displayed in the screen (monitor).

Syntax

```
variable_name = getch();
```

Like getch(), getche() also accepts only single character, but getche() displays the entered character in the screen.

Syntax

```
variable_name = getche();
```

4 **putch(): putch** displays any alphanumeric characters to the standard output device. It displays only one character at a time.

Syntax

```
putch(variable_name);
```

b) String I/O

1.gets() **–** This function is used for accepting any string through stdin (keyboard) until enter key is pressed.

Syntax

```
gets(variable_name);
```

2. puts() – This function prints the string or character array.

Syntax

```
puts(variable_name);
```

3. cgets()- This function reads string from the console.

Syntax

```
cgets(char *st);
```

It requires character pointer as an argument.

4. cputs()- This function displays string on the console.

Syntax


```
cputs(char *st);
```

Example Program

```
void main()
{
char ch[30];
clrscr();
printf("Enter the String : ");
gets(ch);
puts("\n Entered String : %s",ch);
puts(ch);
}
```

Output:

Enter the String : WELCOME

Entered String : WELCOME

Formatted Input & Output Functions

When I/P & O/P is required in a specified format then the standard library functions scanf( ) & printf( ) are used.

O/P function printf( )

The printf( ) function is used to print data of different data types on the console in a specified format.

General Form

```
printf("Control String", var1, var2, …);
```

Control String may contain

1. Ordinary characters
2. Conversion Specifier Field (or) Format Specifiers

They denoted by %, contains conversion codes like %c, %d etc.

and the optional modifiers width, flag, precision, size.

Conversion Codes

Data type	Conversion Symbol
char	%c
int	%d
float	%f
Unsigned octal	%o
Pointer	%p

Width Modifier: It specifies the total number of characters used to display the value.

Precision: It specifies the number of characters used after the decimal point.

E.g: printf(" Number=%7.2\n",5.4321);

Width=7

Precesion=2

Output: Number = 5.43(3 spaces added in front of 5)

Flag: It is used to specify one or more print modifications.

E.g:

Flag	Meaning
-	Left justify the display
+	Display +Ve or –Ve sign of value
Space	Display space if there is no sign
0	Pad with leading 0s

printf("Number=%07.2\n",5.4321)

Output: Number=0005.43

Size: Size modifiers used in printf are,

Size modifier	Converts To
l	Long int
h	Short int
L	Long double

%ld means long int variable

%hd means short int variable

3. Control Codes

They are also known as Escape Sequences.

E.g:

Control Code	Meaning
\n	New line
\t	Horizontal Tab
\b	Back space

Input Function scanf( )

It is used to get data in a specified format. It can accept data of different data types.

Syntax

```
scanf("Control String", var1address, var2address, …);
```

Format String or Control String is enclosed in quotation marks. It may contain

1. White Space
2. Ordinary characters
3. Conversion Specifier Field

It is denoted by % followed by conversion code and other optional modifiers E.g: width, size.

Format Specifiers for scanf( )

Data type	Conversion Symbol
char	%c
int	%d
float	%f
string	%s

E.g Program:


```
#include <stdio.h>
void main( )
{
   int num1, num2, sum;
   printf("Enter two integers: ");
   scanf("%d %d",&num1,&num2);
   sum=num1+num2;
   printf("Sum: %d",sum);
}
```

Output

Enter two integers: 12 11

Sum: 23

Decision making statements

Decision making statements in a programming language help the programmer to transfer the control from one part to other part of the program.

Flow of control: The order in which the program statements are executed is known as flow of control. By default, statements in a c program are executed in a sequential order

Branching statements

Branching statements are used to transfer program control from one point to another.

2 types

a) **Conditional Branching:-** Program control is transferred from one point to another based on the result of some condition

Eg) if, if-else, switch

b) **Unconditional Branching:-** Pprogram control is transferred from one point to another without checking any condition

Eg) goto, break, continue, return

a) Conditional Branching : Selection statements

Statements available in c are

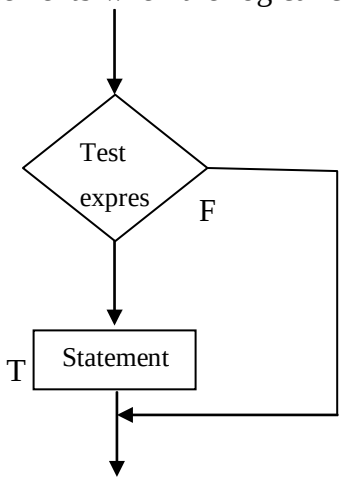
- The **if** statement
- The **if-else** statement
- The **switch case** statement

(i) The if statement

C uses the keyword **if** to execute a statement or set of statements when the logical condition is true.

Syntax:

```
if (test expression)
Statement;
```



- If test expression evaluates to true, the corresponding statement is executed.
- If the test expression evaluates to false, control goes to next executable statement.

- The statement can be single statement or a block of statements. Block of statements must be enclosed in curly braces.

Example:

```
#include <stdio.h>
void main()
{
int n;
clrscr();
printf("enter the number:");
scanf("%d",&n);
if(n>0)
printf("the number is positive");
getch();
}
```

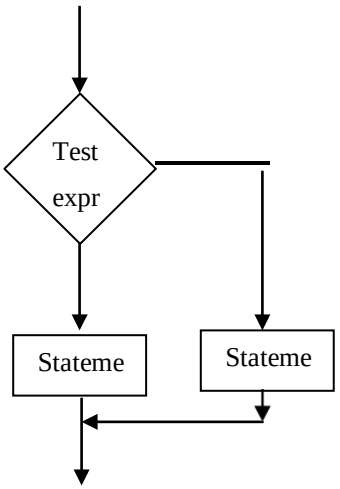
Output:


```
enter the number:50
the number is positive
```

(ii) The if-else statement

Syntax:

```
if (test expression)
Statement T;
else
Statement F;
```



- If the test expression is true, statementT will be executed.
- If the test expression is false, statementF will be executed.
- StatementT and StatementF can be a single or block of statements.

Example:1 Program to check whether a given number is even or odd.

```
#include <stdio.h>
void main(){
int n,r;
clrscr();
printf("Enter a number:");
scanf("%d",&n);
r=n%2;
if(r==0)
printf("The number is even");
else
printf("The number is odd");
getch();
}
```

Output:

```
Enter a number:15
The number is odd
```


Example:2 To check whether the two given numbers are equal

```
void main()
{
int a,b;
clrscr();
printf("Enter a and b:" );
scanf("%d%d",&a,&b);
if(a==b)
printf("a and b are equal");
else
printf("a and b are not equal");
getch();
}
```

Output:

Enter a and b: 2 4

a and b are not equal

(iii) Nested if statement

If the body the if statement contains another if statement, then it is known as nested if statement

Syntax:

```
if (test expression)
        if (test expression)
```

(Or)

```
if (test expression)
{
        statement;
        if (test expression)
        ….
        statement;
}
```

y level.

Syntax:

```
if (test expression1)
{
     …..
    if (test expression2)
     {
      ….
      if (test expression3)
          {
         ….
         if (test expression n)
}}}
```


This is known as if ladder

iv) Nested if-else statement

Here, the if body or else body of an if-else statement contains another if statement or if else statement

Syntax:

if (condition)

{

statement 1;

statement 2;

}

else

```
{
statement 3;
if (condition)
        statementnest;
statement 4;
}
```

Example:

Program for finding greatest of 3 numbers

```
#include<stdio.h>
void main()
{
int a,b,c;
printf("Enter three numbers\n");
scanf("%d%d%d",&a,&b,&c);
if(a>b)
{
        if(a>c)
        {
        printf("a is greatest");
        }
}
else
{       if(b>c)
        {
        printf("b is greatest");
        }
        else
        {
        printf("C is greatest");
        }
}
```

}

Output

Enter three numbers 2 4 6

C is greatest

v) Switch statement

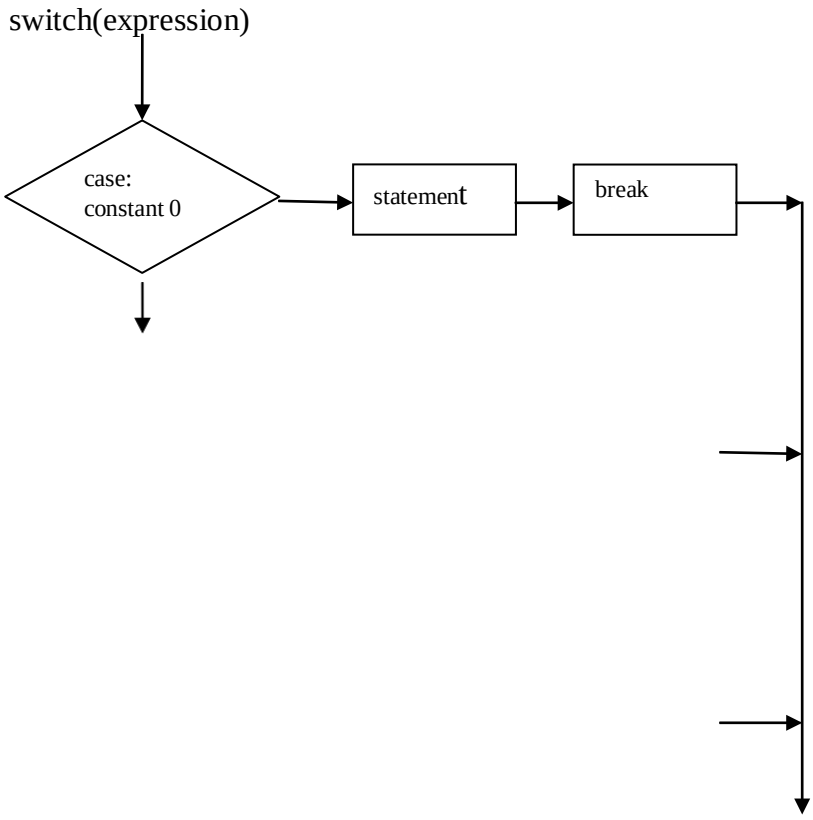
- It is a multi way branch statement.
- It provides an easy & organized way to select among multiple operations depending upon some condition.

Execution

1. Switch expression is evaluated.
2. The result is compared with all the cases.
3. If one of the cases is matched, then all the statements after that matched case gets executed.
4. If no match occurs, then all the statements after the default statement get executed.

- Switch ,case, break and default are keywords
- Break statement is used to exit from the current case structure

Flowchart



Syntax

```
switch(expression)
        {
        case value 1:
                        program statement;
                        program statement;
                        ……
                        break;
        case value 2:
                        program statement;
                        Program statement;
                        ……
                        break;
        …
        case value n:
                        program statement;
                        program statement;
                        ……
                        break;
        default:
                        program statement;
                        program statement;
}
```

End of switch

Example

```
void mai
{
char ch;
printf("Enter a character\n");
scanf("%c",&ch);
switch(ch)
        {
                case „A":
printf("you entered an A\n");
                        break;
case „B":
printf("you entered a B\n");
                        break;
                default:
printf("Illegal entry");
                        break;
        }
 getch();
```

```
}
```

Output

```
Enter a character
A
You entered an A
```

Example2

```
void main()
{
int n;
printf("Enter a number\n");
scanf("%d",&n);
switch(n%2)
        {
                case 0:printf("EVEN\n");
                        break;

case 1:printf("ODD\n");
                        break;
}
getch();
}
```

Output:

```
Enter a number
5
ODD
```

b)Unconditional branching statements

i)The goto Statement

This statement does not require any condition. Here program control is transferred to another part of the program without testing any condition.

Syntax:

```
goto label;
```

- label is the position where the control is to be transferred.

Example:

```
void main()
{
printf("www.");
goto x;
y:
printf("mail");
goto z;
x:
printf("yahoo");
goto y;
z:
printf(".com");
getch();
}
```


Output : www.yahoomail.com

b) break statement

- A break statement can appear only inside a body of , a switch or a loop
- A break statement terminates the execution of the nearest enclosing loop or switch.

Syntax

```
break;
```

Example:1

```
#include<stdio.h>
void main()
{
```

```
int c=1;
while(c<=5)
{
        if (c==3)
          break;
printf("\t %d",c);
c++;
}
}
```

Output : 1 2

Example :2

```
#include<stdio.h>
void main()
{
int i;
for(i=0;i<=10;i++)
{
  if (i==5)
    break;
  printf(" %d",i);
}
}
```


Output :1 2 3 4

iii) continue statement

- A continue statement can appear only inside a loop.
- A continue statement terminates the current iteration of the nearest enclosing loop.

Syntax:

```
continue;
```

Example :1

```
#include<stdio.h>
void main()
{
int c=1;
while(c<=5)
{
  if (c==3)
    continue;
  printf("\t %d",c);
  c++;
}
}
```

Output : 1 2 4 5

Example :2

```
#include<stdio.h>
main()
{
int i;
for(i=0;i<=10;i++)
{
  if (i==5)
    continue;
  printf(" %d",i);
}
}
```

Output :1 2 3 4 6 7 8 9 10

iv) return statement:

A return statement terminates the execution of a function and returns the control to the calling function.

Syntax:

```
return;
```

(or)

```
return expression;
```

Looping statements

Iteration is a process of repeating the same set of statements again and again until the condition holds true.

Iteration or Looping statements in C are:

1. for
2. while
3. do while

Loops are classified as

- Counter controlled loops
- Sentinel controlled loops

Counter controlled loops

- The number of iterations is known in advance. They use a control variable called loop counter.
- Also called as definite repetitions loop.
- E.g: for

Sentinel controlled loops

- The number of iterations is not known in advance. The execution or termination of the loop depends upon a special value called sentinel value.
- Also called as indefinite repetitions loop.
- E.g: while

1. for loop

It is the most popular looping statement.

Syntax:

```
for(initialization;condition2;incrementing/updating)
{
        Statements;
}
```

There are three sections in for loop.

a. **Initialization section –** gives initial value to the loop counter.
b. **Condition section –** tests the value of loop counter to decide whether to execute the loop or not.
c. **Manipulation section -** manipulates the value of loop counter.

Execution of for loop

1. Initialization section is executed only once.
2. Condition is evaluated
    a. If it is true, loop body is executed.
    b. If it is false, loop is terminated
3. After the execution of loop, the manipulation expression is evaluated.
4. Steps 2 & 3 are repeated until step 2 condition becomes false.

Ex 1: Write a program to print 10 numbers using for loop

```
void main()
{
int i;
clrscr();
for(i=1;i<=10;i++)
{
printf("%d",i);
}
getch();
}
```

Output:

1 2 3 4 5 6 7 8 9 10

Ex2: To find the sum of n natural number. 1+2+3…+n

```
void main()
{
int n, i, sum=0;
clrscr();
printf("Enter the value for n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
sum=sum+i;
}
printf("Sum=%d", sum);
getch();
}
```

Output:

```
Enter the value for n
4
sum=10
```


2. while statement

- They are also known as Entry controlled loops because here the condition is checked before the execution of loop body.

Syntax:

```
while (expression)
{
statements;
}
```

Execution of while loop

a. Condition in while is evaluated
   i. If it is true, body of the loop is executed.

ii. If it is false, loop is terminated

b. After executing the while body, program control returns back to while header.

c. Steps a & b are repeated until condition evaluates to false.

d. Always remember to initialize the loop counter before while and manipulate loop counter inside the body of while.

Ex 1: Write a program to print 10 numbers using while loop

```
void main()
{
int i=1;
clrscr();
while (num<=10)
{
printf (“%d”,i);
i=i+1;
}
getch();
}
```


Output:

1 2 3 4 5 6 7 8 9 10

Ex2: To find the sum of n natural number. 1+2+3…+n

```
void main()
{
int n, i=1, sum=0;
clrscr();
printf(“Enter the value for n”);
scanf(“%d”,&n);
while(i<=n)
{
sum=sum+i;
```

```
i=i+1;
}
printf("Sum=%d", sum);
getch();
}
```

Output:

Enter the value for n

4

Sum=10

3. do while statement

- They are also known as Exit controlled loops because here the condition is checked after the execution of loop body.

Syntax:

```
do
{
statements;
}
while(expression);
```


Execution of do while loop

a. Body of do-while is executed.
b. After execution of do-while body, do-while condition is evaluated
    i. If it is true, loop body is executed again and step b is repeated.
    ii. If it is false, loop is terminated

- The body of do-while is executed once, even when the condition is initially false.

Ex1: Write a program to print 10 numbers using do while loop

```
void main()
```

```
{
int  i=1;
clrscr();
do
{
printf ("%d",i);
i=i+1;
} while (num<=10);
getch();
}
```

Output:

1 2 3 4 5 6 7 8 9 10

Ex2: To find the sum of n natural number. 1+2+3…+n

```
void main()
{
int n, i=1, sum=0;
clrscr();
printf("Enter the value for n");
scanf("%d",&n);
do
{
sum=sum+i;
i=i+1;
} while(i<=n);
printf("Sum=%d", sum);
getch();
}
```


Output:

Enter the value for n

4

Sum=10

Three main ingredients of counter-controlled looping

1. Initialization of loop counter
2. Condition to decide whether to execute loop or not.
3. Expression that manipulates the value of loop.

Nested loops

- If the body of a loop contains another iteration statement, then we say that the loops are nested.
- Loops within a loop are known as nested loop.

Syntax

```
while(condition)
{
        while(condition)
        {
        Statements;
        }
Statements;
}
```

UNIT IIARRAYS AND STRINGS

Introduction to Arrays: Declaration, Initialization – One dimensional array – Example Program: Computing Mean, Median and Mode - Two dimensional arrays – Example Program: Matrix Operations (Addition, Scaling, Determinant and Transpose) - String operations: length, compare, concatenate, copy – Selection sort, linear and binary search

Introduction to Arrays

An Array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as subscript). Subscript indicates an ordinal number of the elements counted from the beginning of the array.

Definition:

An array is a data structure that is used to store data of the same type. The position of an element is specified with an integer value known as **index** or **subscript**.

E.g.

1	3	5	2

a(integer array)

1.2	3.5	5.4	2.1

b(float array)

[0] [1] [2] [3]

Characteristics:

i) All the elements of an array share a common name called as array name

ii) The individual elements of an array are referred based on their position.

iii) The array index in c starts with 0.

In general arrays are classified as:

1. Single dimensional array
2. Multi-dimensional array

Declarations of Arrays

Array has to be declared before using it in C Program. Declaring Array means specifying three things.

Data_type Data Type of Each Element of the array

Array_name Valid variable name

Size Dimensions of the Array

Arrays are declared using the following syntax:

type name[size]

Here the type can be either int, float, double, char or any oher valid data type. The number within the brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array.

ex: int marks[10]

Initialization of arrays

Elements of the array can also be initialized at the time of declaration as in the case of every other variable. When an array is initialized, we need to provide a value for every element in the array. Arrays are initialized by writing,

type array_name[size] = { list of values};

The values are written with curly brackets and every value is separated by a comma. It is a compiler error to specify more number of values than the number of elements in the array.

ex: int marks[5] = {90, 92, 78, 82, 58};

One dimensional Array

- It is also known as one-dimensional arrays or linear array or vectors
- It consists of fixed number of elements of same type
- Elements can be accessed by using a single subscript. eg) a[2]=9;

Eg)

1	3	5	2

a

[0] [1] [2] [3] ← subscripts or indices

Declaration of Single Dimensional Array

Syntax:

```
datatype  arrayname [array size];
```

E.g. int a[4]; // a is an array of 4 integers

char b[6]; //b is an array of 6 characters

Initialization of single dimensional array

Elements of an array can also be initialized.

Rules

a) Elements of an array can be initialized by using an initialization list. An initialization list is a comma separated list of initializers enclosed within braces.

Eg) int a[3]={1,3,4};

b) If the number of initializers in the list is less than array size, the leading array locations gets initialized with the given values. The rest of the array locations gets initialized to

0 - for int array

0.0 - for float array

\0 - for character array

Eg) int a[2]={1};

1	0

a

char b[5]={'A'.'r','r'};

b

'A'	'r'	'r'	'\0'	'\0'

Usage of single dimensional array

The elements of single dimensional array can be accessed by using a subscript operator([]) and a subscript.

Reading storing and accessing elements:

An iteration statement (i.e loop) is used for storing and reading elements.

Ex:1 Program to calculate the average marks of the class

```
#include <stdio.h>
void main()
{
        int m[5],i,sum=0,n;
        float avg;
        printf("enter number of students \n");
        scanf("%d",&n);
        printf("enter marks of students \n");
        for(i=0;i<n;i++)
        {
                scanf("%d",&m[i]);
        }
        for(i=0;i<n;i++)
        sum=sum+m[i];
        avg=(float)sum/n;
        printf("average=%f",avg);
}
```


Output:

Enter number of students

5

Enter marks of students

55

60

78

85

90

Average=73.6

Example Programs

C Program to Find Mean, Median, and Mode of Given Numbers.

```
#define SIZE 100
#include"stdio.h"
float mean_function(float[],int);
float median_function(float[],int);
float mode_function(float[],int);
int main()
{
int i,n,choice;
float array[SIZE],mean,median,mode;
printf("Enter No of Elements\n");
scanf("%d",&n);
printf("Enter Elements\n");
for(i=0;i
scanf("%f",&array[i]);
do
{
printf("\n\tEnter Choice\n\t1.Mean\n\t2.Median\n\t3.Mode\n4.Exit");
scanf("%d",&choice);
switch(choice)
{
case 1: mean=mean_function(array,n);
printf("\n\tMean = %f\n",mean);
break;
```

```
case 2: median=median_function(array,n);
printf("\n\tMedian = %f\n",median);
break;
case 3: mode=mode_function(array,n);
printf("\n\tMode = %f\n",mode);
break;
case 4: break;
default:printf("Wrong Option");
break;
}
}while(choice!=4);
}
float mean_function(float array[],int n)
{
int i;
float sum=0;
for(i=0;i
sum=sum+array[i];
return (sum/n);
}
float median_function(float a[],int n)
{
float temp;
int i,j;
for(i=0;i
for(j=i+1;j
{
```

```
if(a[i]>a[j])
{
temp=a[j];
a[j]=a[i];
a[i]=temp;
}
}
if(n%2==0)
return (a[n/2]+a[n/2-1])/2;
else
return a[n/2];
}
float mode_function(float a[],int n)
{
return (3*median_function(a,n)-2*mean_function(a,n));
}
```


Output

```
Enter Elements
2
3
4

Enter Choice
1.Mean
2.Median
3.Mode
4.Exit
```

1

Mean = 3.000000

Enter Choice

1.Mean

2.Median

3.Mode

4.Exit

2

Median = 3.000000

Enter Choice

1.Mean

2.Median

3.Mode

4.Exit

3

Mode = 3.000000

Enter Choice

1.Mean

2.Median

3.Mode

4.Exit

4

Two dimensional Array

- A 2D array is an array of 1-D arrays and can be visualized as a plane that has rows and columns.
- The elements can be accessed by using two subscripts, row subscript (row no), column subscript(column no).
- It is also known as matrix.

E.g,

a[3][5]

1	2	3	6	7
9	10	5	0	4
3	1	2	1	6

Declaration

datatype arrayname [row size][column size]

e.g) int a [2][3]; //a is an integer array of 2 rows and 3 columns

number of elements=2*3=6

Initialization

1. By using an initialization list, 2D array can be initialized.

e.g. int a[2][3] = {1,4,6,2}

1	4	6
2	0	0

a

2. The initializers in the list can be braced row wise.

e.g int a[2][3] = {{1,4,6} , {2}};

Example Programs

Progarm for addition,transpose and multiplication of array

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,i,k,j,c1,c2,r1,r2;
        int m1[10][10],m2[10][10],m3[10][10];
        clrscr();
        while(1)
        {

                printf("\n 1. Transpose of Matrix:-\n");
                printf("\n 2. Addition of Matrix:-\n");
                printf("\n 3. Multiplication of Matrix:-\n");
                printf("\n 4. Exit\n");
                printf("\n Enter your choice:-");
                scanf("%d",&a);
                switch(a)
                {
                case 1 :
                        printf("\n Enter the number of row and coloum:-");
                        scanf("%d%d",&r1,&c1);
                        printf("\n Enter the element :-");
                        for(i=0;i<r1;i++)
                        {
                                for(j=0;j<c1;j++)
```

```
            {
                scanf("%d",&m1[i][j]);
                m2[j][i]=m1[i][j];
            }
        }
        /*Displaying transpose of matrix*/
        printf("\n Transpose of Matrix is:-\n");
        for(i=0;i<r1;i++)
        {
            for(j=0;j<c1;j++)
                printf("\t%d",m2[i][j]);
            printf("\n");
        }
        break;
```



```
case 2:
        printf("\n how many row and coloum in Matrix one:-");
        scanf("%d%d",&r1,&c1);
        printf("\n How amny row and coloum in Matrix two:-");
        scanf("%d%d",&r2,&c2);
        if((r1==r2)&&(c1==c2))
        {
            printf("\n Addition is possible:-");
            printf("\n Input Matrix one:-");
            for(i=0;i<r1;i++)
            {
                for(j=0;j<c1;j++)
                    scanf("%d",&m1[i][j]);
```

```
                }
                printf("\n Input Matrix two:-");
                for(i=0;i<r2;i++)
                {
                        for(j=0;j<c2;j++)
                                scanf("%d",&m2[i][j]);
                }
                /* Addition of Matrix*/
                for(i=0;i<r1;i++)
                {
                        for(j=0;j<c1;j++)
                                m3[i][j]=m1[i][j]+ m2[i][j];
                }
                printf("\n The sum is:-\n");
                for(i=0;i<c1;i++)
                {
                        for(j=0;j<r1;j++)
                                printf("%5d",m3[i][j]);
                        printf("\n");
                }
        }
        else
                printf("\n Addition is not possible:-");

        break;
case 3:
```

```
printf("\n Enter number of row and coloum in matrix one:-");
scanf("%d%d",&r1,&c1);
printf("\n Enter number of row and coloum in matrix two:-");
scanf("%d%d",&r2,&c2);
if(c1==r2)
{
        printf("\n Multiplication is possible:-");
        printf("\n Input value of Matrix one:-");
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c1;j++)
                        scanf("%d",&m1[i][j]);
        }
        printf("\n Input value of Matrix two:-");
        for(i=0;i<r2;i++)
        {
                for(j=0;j<c2;j++)
                        scanf("%d",&m2[i][j]);
        }
        for(i=0;i<r1;i++)
                for(j=0;j<c2;j++)
                {
                        m3[i][j]=0;
                        for(k=0;k<c1;k++)
                                m3[i][j]=m3[i][j]+m1[i][k]*m2[k][j];
                }
                /*Displaying final matrix*/
```

```
                printf("\n Multiplication of Matrix:-\n");
                for(i=0;i<r1;i++)
                {
                        for(j=0;j<c2;j++)
                                printf("\t%d",m3[i][j]);
                        printf("\n");
                }
        }
        else
                printf("\n Multiplication is not possible");

        break;
    case 4:
        exit(0);
        break;
    }
    getch();
  }
}
```


String Operations

Definition:

The group of characters, digits, & symbols enclosed within double quotes is called as Strings. Every string is terminated with the NULL (‘\0’) character.

E.g. “INDIA” is a string. Each character of string occupies 1 byte of memory. The last character is always ‘\0’.

Declaration:

String is always declared as character arrays.

Syntax

```
char stringname[size];
```

E.g. char a[20];

Initialization:

We can use 2 ways for initializing.

1. By using string constant

E.g. char str[6]= “Hello”;

2. By using initialisation list

E.g. char str[6]={‘H’, ‘e’, ‘l’, ;l’, ;o’, ‘\0’};

String Operations or String Functions

These functions are defined in **string.h** header file.

1. strlen() function

It is used to find the length of a string. The terminating character (‘\0’) is not counted.

Syntax

```
temp_variable = strlen(string_name);
```

E.g.

s= “hai”;

strlen(s)-> returns the length of string s i.e. 3.

2. strcpy() function

It copies the source string to the destination string

Syntax

```
strcpy(destination,source);
```

E.g.

s1="hai";

s2= "welcome";

strcpy(s1,s2); -> s2 is copied to s1. i.e. s1=welcome.

3. strcat() function

It concatenates a second string to the end of the first string.

Syntax

```
strcat(firststring,    secondstring);
```

E.g.

s1="hai ";

s2= "welcome";

strcat(s1,s2); -> s2 is joined with s1. Now s1 is hai welcome.

E.g. Program:

```
        #include <stdio.h>
#include <string.h>
void main ()
{
  char str1[20] = "Hello";
  char str2[20] = "World";
  char str3[20];
  int len ;
  strcpy(str3, str1);
  printf("Copied String=  %s\n", str3 );
```

```
   strcat( str1, str2);
   printf("Concatenated String is= %s\n", str1 );
   len = strlen(str1);
   printf("Length of string str1 is= %d\n", len );
   return 0;
}
```

Output:

Copied String=Hello

Concatenated String is=HelloWorld

Length of string str1is

4. **strcmp() function**

It is used to compare 2 strings.

Syntax


```
temp_varaible=strcmp(string1,string2)
```

- If the first string is greater than the second string a positive number is returned.
- If the first string is less than the second string a negative number is returned.
- If the first and the second string are equal 0 is returned.

5. **strlwr() function**

It converts all the uppercase characters in that string to lowercase characters.

Syntax

```
strlwr(string_name);
```

E.g.

```
str[10]= “HELLO”;
strlwr(str);
puts(str);
```

Output: hello

6. strupr() function

It converts all the lowercase characters in that string to uppercase characters.

Syntax

```
strupr(string_name);
```

E.g.

```
str[10]= “HEllo”;
strupr(str);
puts(str);
```


Output: HELLO

7. strrev() function

It is used to reverse the string.

Syntax

```
strrev(string_name);
```

E.g.

```
str[10]= “HELLO”;
strrev(str);
puts(str);
```

Output: OLLEH

String functions

Functions	Descriptions
strlen()	Determines the length of a String
strcpy()	Copies a String from source to destination
strcmp()	Compares two strings
strlwr()	Converts uppercase characters to lowercase
strupr()	Converts lowercase characters to uppercase
strdup()	Duplicates a String
strstr()	Determines the first occurrence of a given String in another string
strcat()	Appends source string to destination string
strrev()	Reverses all characters of a string

Example: String Comparison


```
void main()
{
char s1[20],s2[20];
int val;
printf("Enter String 1\n");
gets(s1);
printf("Enter String 2\n");
gets (s2);
val=strcmp(s1,s2);
if (val==0)
printf("Two Strings are equal");
else
printf("Two Strings are not equal");
getch();
```

}

Output:

Enter String 1

Computer

Enter String 2

Programming

Two Strings are not equal

2.8.1 String Arrays

They are used to store multiple strings. 2-D char array is used for string arrays.

Declaration

```
char arrayname[rowsize][colsize];
```


E.g.

char s[2][30];

Here, s can store 2 strings of maximum 30 characters each.

Initialization

2 ways

1. Using string constants

```
char s[2][20]={“Ram”, “Sam”};
```

2. Using initialization list.

```
char s[2][20]={ {‘R’, ‘a’, ‘m’, ‘\0’},
                {‘S’, ‘a’, ‘m’, ‘\0’}};
```

E.g. Program

```
#include<stdio.h>
void main()
{
```

```
int i;
char s[3][20];
printf("Enter Names\n");
for(i=0;i<3;i++)
scanf("%s", s[i]);
printf("Student Names\n");
for(i=0;i<3;i++)
printf("%s", s[i]);
}
```

Sorting

Sorting is the process of arranging elements either in ascending or in descending order.

Sorting Methods

1. Selection Sort
2. Bubble Sort
3. Merge sort
4. Quick sort

1. Selection sort

It finds the smallest element in the list & swaps it with the element present at the head of the list.

E.g.

25 20 15 10 5

5 20 15 10 25

5 10 15 20 25

2. Bubble Sort

In this method, each data item is compared with its neighbour element. If they are not in order, elements are exchanged.

With each pass, the largest of the list is "bubbled" to the end of the list.

E.g.

Pass 1:

25 20 15 10 5

20 25 15 10 5

20 15 25 10 5

20 15 10 25 5

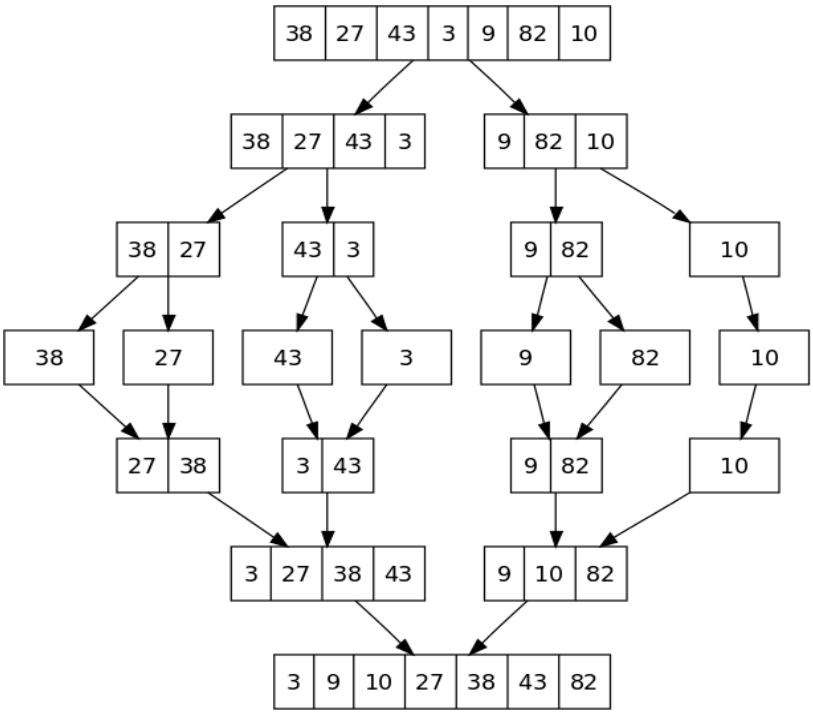
20 15 10 5 25

25 is the largest element

Repeat same steps until the list is sorted

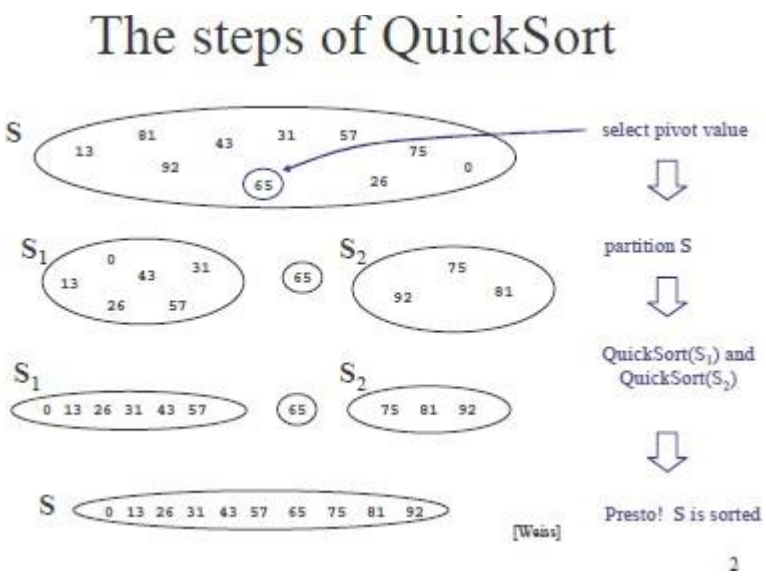
3. Merge Sort:

- Merge sort is based on Divide and conquer method.
- It takes the list to be sorted and divide it in half to create two unsorted lists.
- The two unsorted lists are then sorted and merged to get a sorted list.



4. Quick Sort

- This method also uses the technique of 'divide and conquer'.
- Pivot element is selected from the list, it partitions the rest of the list into two parts – a sub-list that contains elements less than the pivot and other sub-list containing elements greater than the pivot.
- The pivot is inserted between the two sub-lists. The algorithm is recursively applied to sort the elements.



Program:

```
#include <stdio.h>
void main()
{
   int i, j, temp, n, a[10];
   printf("Enter the value of N \n");
   scanf("%d", &n);
   printf("Enter the numbers \n");
   for (i = 0; i < n; i++)
      scanf("%d", &a[i]);
   for (i = 0; i < n; i++)
   {
      for (j = i + 1; j < n; j++)
      {
         if (a[i] > a[j])
         {
            temp =  a[i];
```

```
            a[i] = a[j];
            a[j] = temp;
        }
    }
}
printf("The numbers arranged in ascending order are given below \n");
for (i = 0; i < n; i++)
    printf("%d\n", a[i]);
printf("The numbers arranged in descending order are given below \n");
for(i=n-1;i>=0;i--)
    printf("%d\n",a[i]);
}
```

Output:


```
Enter the value of N
4
Enter the numbers
10 2 5 3
The numbers arranged in ascending order are given below
2
3
5
10
The numbers arranged in descending order are given below
10
5
3
2
```

Searching

Searching is an operation in which a given list is searched for a particular value. If the value is found its position is returned.

Types:

1. Linear Search
2. Binary Search

1. Linear Search

The search is linear. The search starts from the first element & continues in a sequential fashion till the end of the list is reached. It is slower method.

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a[10],i,n,m,c=0;
   clrscr();
   printf("Enter the size of an array: ");
   scanf("%d",&n);
   printf("Enter the elements of the array: ");
   for(i=0;i<=n-1;i++)
   scanf("%d",&a[i]);
   printf("Enter the number to be searched: ");
   scanf("%d",&m);
   for(i=0;i<=n-1;i++)
   {
      if(a[i]==m)
      {
```

```
            printf("Element is in the position %d\n",i+1);
            c=1;
            break;
        }
    }
    if(c==0)
        printf("The number is not in the list");
    getch();
}
```

Output:

Enter the size of an array: 4

Enter the elements of the array: 4 3 5 1

Enter the number to be search: 5

Element is in the position 3

2. Binary Search

- If a list is already sorted then we can easily find the element using binary serach.
- It uses divide and conquer technique.

Steps:

1. The middle element is tested with searching element. If found, its position is returned.
2. Else, if searching element is less than middle element, search the left half else search the right half.
3. Repeat step 1 & 2.

Program:

```
#include<stdio.h>
void main()
{
   int a[10],i,n,m,c=0,l,u,mid;
   printf("Enter the size of an array: ");
   scanf("%d",&n);
   printf("Enter the elements in ascending order: ");
   for(i=0;i<n;i++)
       scanf("%d",&a[i]);
   printf("Enter the number to be searched: ");
   scanf("%d",&m);
   l=0,u=n-1;
   while(l<=u)
  {
       mid=(l+u)/2;
       if(m==a[mid])
{
          c=1;
          break;
         }
       else if(m<a[mid])
{
          u=mid-1;
        }
       else
          l=mid+1;
   }
```

```
    if(c==0)
        printf("The number is not found.");
    else
        printf("The number is found.");
}
```

Sample output:

Enter the size of an array: 5

Enter the elements in ascending order: 4 7 8 11 21

Enter the number to be search: 11

The number is found.

Example:

3 5 7 9 11

Search key=7 middle element=7

Searching element=middle element. So the element is found.

Search key=11

Middle element=7

Searching element>middle

So go to right half: 9 11. Repeat steps until 11 is found or list ends.

UNIT III FUNCTIONS AND POINTERS

Introduction to functions: Function prototype, function definition, function call, Built-in functions (string functions, math functions) – Recursion – Example Program: Computation of Sine series, Scientific calculator using built-in functions, Binary Search using recursive functions – Pointers – Pointer operators – Pointer arithmetic – Arrays and pointers – Array of pointers – Example Program: Sorting of names – Parameter passing: Pass by value, Pass by reference – Example Program: Swapping of two numbers and changing the value of a variable using pass by reference

Introduction to functions

A function is a subprogram of one or more statements that performs a specific task when called.

Advantages of Functions:

1. Code reusability
2. Better readability
3. Reduction in code redundancy
4. Easy to debug & test.

Classification of functions:

- Based on who develops the function
- Based on the number of arguments a function accepts

1. Based on who develops the function

There are two types.

1. Library functions
2. User-defined functions

1. Library functions [Built-in functions]

Library functions are predefined functions. These functions are already developed by someone and are available to the user for use. Ex. printf( ), scanf( ).

2. User-defined functions

User-defined functions are defined by the user at the time of writing a program. Ex. sum( ), square( )

Using Functions

A function can be compared to a *black box* that takes in inputs, processes it, and then outputs the

result. Terminologies using functions are:

- A function *f* that uses another function *g* is known as the *calling function*, and *g* is known as the *called function*.
- The inputs that a function takes are known as *arguments.*
- When a called function returns some result back to the calling function, it is said to *return* that result.
- The calling function may or may not pass *parameters* to the called function. If the called function accepts arguments, the calling function will pass parameters, else not.
- *Function declaration* is a declaration statement that identifies a function's name, a list of arguments that it accepts, and the type of data it returns.
- *Function definition* consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function.

Function Prototype

Before using a function, the compiler must know the number of parameters and the type of parameters that the function expects to receive and the data type of

value that it will return to the calling program. Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function**.**

Syntax:

return_data_type function_name(data_type variable1, data_type variable2,..);

Here, function_name is a valid name for the function. Naming a function follows the same rules that are followed while naming variables. A function should have a meaningful name that must specify the task that the function will perform.

return_data_type specifies the data type of the value that will be returned to the calling function as a result of the processing performed by the called function.

(data_type variable1, data_type variable2, ...) is a list of variables of specified data types.

These variables are passed from the calling function to the called function. They are also known as arguments or parameters that the called function accepts to perform its task.

Function definition

When a function is defined, space is allocated for that function in the memory. A function definition comprises of two parts:

- Function header
- Function body

The syntax of a function definition can be given as:

return_data_type function_name(data_type variable1, data_type variable2,..)

{

.....

statements

.....

return(variable);

}

While return_data_type function_name(data_type variable1, data_type variable2,...) is known as the function header, the rest of the portion comprising of program statements within the curly brackets { } is the function body which contains the code to perform the specific task.

Note that the function header is same as the function declaration. The only difference between the two is that a function header is not followed by a semi-colon.

Function Call

The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function.

Syntax:

function_name(variable1, variable2, ...);

The following points are to be noted while calling a function:

- Function name and the number and the type of arguments in the function call must be same as that given in the function declaration and the function header of the function definition.
- Names (and not the types) of variables in function declaration, function call, and header of function definition may vary.
- Arguments may be passed in the form of expressions to the called function. In such a case, arguments are first evaluated and converted to the type of formal parameter and then the body of the function gets executed.

- If the return type of the function is not void, then the value returned by the called function may be assigned to some variable as given below.
  variable_name = function_name(variable1, variable2, ...);

Working of a function

```
void main()
{
int x,y,z;
int abc(int, int, int)  // Function declaration
…..
…..
abc(x,y,z) // Function Call
…       Actual arguments
…
}

int abc(int i, int j, int k)   // Function definition
{               Formal arguments
…….
….
return (value);
}
```


Calling function – The function that calls a function is known as a calling function.

Called function – The function that has been called is known as a called function.

Actual arguments – The arguments of the calling function are called as actual arguments.

Formal arguments – The arguments of called function are called as formal arguments.

Steps for function Execution:

1. After the execution of the function call statement, the program control is transferred to the called function.
2. The execution of the calling function is suspended and the called function starts execution.
3. After the execution of the called function, the program control returns to the calling function and the calling function resumes its execution.

Built-in functions (string functions, math functions)

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc. These functions are defined in the header file**.**

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file.

There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

Library of Mathematical functions.

These are defined in math.h header file.

Example:

1	double cos(double x)- Returns the cosine of a radian angle x
2	double sin(double x)- Returns the sine of a radian angle x.
3	double exp(double x)- Returns the value of e raised to the x^{th} power
4	double log(double x) Returns the natural logarithm (base-e logarithm) of x.
5	double sqrt(double x) Returns the square root of x.
6	double pow(double x, double y) Returns x raised to the power of y.

Library of standard input & output functions

Header file: stdio.h

Example:

1	printf()	This function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen
2	scanf()	This function is used to read a character, string, and numeric data from keyboard.
3	getc()	It reads character from file
4	gets()	It reads line from keyboard

Library of String functions:

Header file: string.h

Example:

Functions	Descriptions
strlen()	Determines the length of a String
strcpy()	Copies a String from source to destination
strcmp()	Compares two strings

strlwr()	Converts uppercase characters to lowercase
strupr()	Converts lowercase characters to uppercase

Example: strlen function

```
#include <stdio.h>
int main() {
char string1[20];
char string2[20];
strcpy(string1, "Hello");
strcpy(string2, "Hellooo");
printf("Length of string1 : %d\n", strlen( string1 ));
printf("Length of string2 : %d\n", strlen( string2 ));
return 0;
}
```


Output:

```
Length of string1 : 5
Length of string2 : 7
```

Example: strcpy function

```
#include <stdio.h>
int main() {
  char input_str[20];
  char *output_str;
  strcpy(input_str, "Hello");
  printf("input_str: %s\n", input_str);
  output_str = strcpy(input_str, "World");
```

```
  printf("input_str: %s\n", input_str);
  printf("output_str: %s\n", output_str);
  return 0;
}
```

Output:

input_str: Hello

input_str: World

output_str: World

Example :strcmp function

```
#include<stdio.h>
#include<string.h>
void main( )
{
char one[20] = “William Lambton”;
char two[20] = “William Lamberton”;
if(strcmp(one, two) == 0)
printf(“The names are the same.”);
else
printf(“The names are different.”);
}
```


Output:

The names are different

Example: Strupr() and strlwr()

```
#include<stdio.h>
#include<string.h>
int main( )
{
char str[] = " String Functions";
printf("%s \n", strupr(str));
printf("%s \n", strlwr(str));
return 0;
}
```

Output:

STRING FUNCTIONS

string functions

Recursion

A function that calls itself is known as a recursive function.

Direct & Indirect Recursion:

Direct Recursion:

A function is directly recursive if it calls itself.

```
A( )
{
….
….
A( );// call to itself
….
}
```

Indirect Recursion:

Function calls another function, which in turn calls the original function.

```
A( )
{
…
…
B( );
…
}
B( )
{
…
…
A( );// function B calls A
…
}
```


Consider the calculation of 6! (6 factorial)

ie 6! = 6 * 5 * 4 * 3 * 2 * 1

6! = 6 * 5!

6! = 6 * (6 - 1)!

n! = n * (n - 1)!

Types of Recursion

Direct Recursion

A function is said to be *directly* recursive if it explicitly calls itself. Here, the function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.

```
int Func (int n)
{
```

```
if (n == 0)
return n;
else
return (Func (n–1));
}
```

Indirect Recursion

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. These two functions are indirectly recursive as they both call each other.

```
int Funcl (int n)
{
if (n == 0)
return n;
else
return Func2(n);
}
int Func2(int x)
{
return Func1(x–1);
}
```


Tail Recursion

A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller. When the called function returns, the returned value is immediately returned from the calling function.

```
int Fact(int n)
{
if (n == 1)
return 1;
else
return (n * Fact(n–1));
}
```

The above function is a nontail-recursive function, because there is a pending operation of multiplication to be performed on return from each recursive call. Whenever there is a pending operation to be performed, the function becomes non-tail recursive. In such a non-tail recursive function, information about each pending operation must be stored, so the amount of information directly depends on the number of calls.


```
int Fact(n)
{
return Fact1(n, 1);
}
int Fact1(int n, int res)
{
if (n == 1)
return res;
else
return Fact1(n–1, n*res);
}
```

The same factorial function can be written in a tail recursive manner. In the code, Fact1 function preserves the syntax of Fact(n). Here the recursion occurs in the Fact1 function and not in Fact function. Fact1 has no pending operation to be

performed on return from recursive calls. The value computed by the recursive call is simply returned without any modification. So in this case, the amount of information to be stored on the system stack is constant (only the values of n and res need to be stored) and is independent of the number of recursive calls.

E.g. Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int fact(int);
int n,f;
printf("Enter the number \n");
scanf("%d",&n);
f=fact(n);
printf("The factorial of a number =%d",f);
getch();
}
int fact(int n)
{
if(n==1)
return(1);
else
return n*fact(n-1);
}
```


OUTPUT

Enter the number to find the factorial

5

The factorial of a number=120

Pattern of Recursive Calls:

Based on the number of recursive calls, the recursion is classified in to 3 types. They are,

1. Linear Recursion - Makes only one recursive call.
2. Binary Recursion - Calls itself twice.
3. N-ary recursion - Calls itself n times.

Converting Recursive Functions to Tail Recursive

A non-tail recursive function can be converted into a tail-recursive function by using an *auxiliary parameter* as we did in case of the Factorial function. The auxiliary parameter is used to form the result. When we use such a parameter, the pending operation is incorporated into the auxiliary parameter so that the recursive call no longer has a pending operation.

Recursive functions can also be characterized depending on the way in which the recursion grows in a linear fashion or forming a tree structure as shown below:

```
int Fibonacci(int num)
{
if(num == )
return  ;
else
return (Fibonacci(num - 1) + Fibonacci(num – 2));
}
else if (num == 1)
```

return 1;

Observe the series of function calls. When the function pending operations in turn calls the function

Fibonacci(7) = Fibonacci(6) + Fibonacci(5)

Fibonacci(6) = Fibonacci(5) + Fibonacci(4)

Fibonacci(5) = Fibonacci(4) + Fibonacci(3)

Fibonacci(4) = Fibonacci(3) + Fibonacci(2)

Fibonacci(3) = Fibonacci(2) + Fibonacci(1)

Fibonacci(2) = Fibonacci(1) + Fibonacci(0)

Now we have, Fibonacci(2) = 1 + 0 = 1

Fibonacci(4) = 2 + 1 = 3

Fibonacci(5) = 3 + 2 = 5

Fibonacci(6) = 3 + 5 = 8

Fibonacci(7) = 5 + 8 = 13

On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function. For example, the Fibonacci function in which the pending operations recursively call the Fibonacci function.

Tower of Hanoi

The tower of Hanoi is one of the main applications of recursion. It says, 'if you can solve n–1 cases, then you can easily solve the nth case'. The figure (a) below shows three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk.

In our case, A is the source pole, C is the destination pole, and B is the spare pole. To transfer all the three rings from A to C, we will first shift the upper two rings (n–1 rings) from the source pole to the spare pole. We move the first two rings from pole A to B as shown in figure (b) .

Now that n–1 rings have been removed from pole A, the nth ring can be easily moved from the source pole (A) to the destination pole (C). Figure (c) shows this step.

The final step is to move the n–1 rings from the spare pole (B) to the destination pole (C). This is shown in Fig. (d)

To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

Base case: if n=1

- Move the ring from A to C using B as spare

Recursive case:

- Move n – 1 rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move n – 1 rings from B to C using A as spare

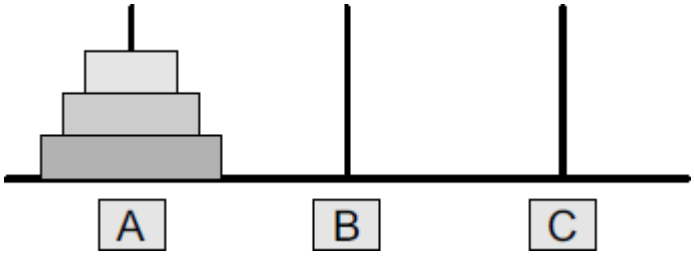


Figure (a)

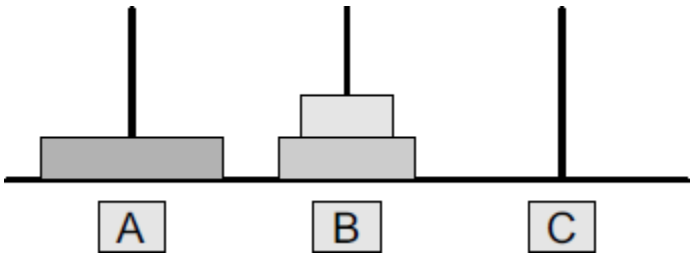


Figure (b)

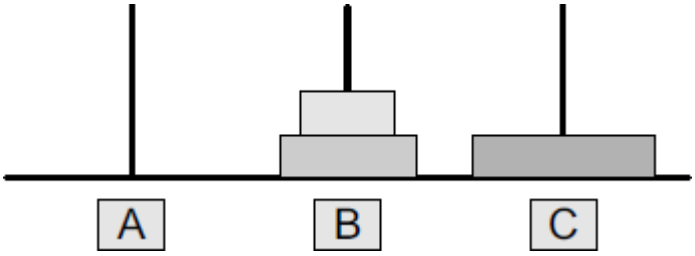


Figure (c)

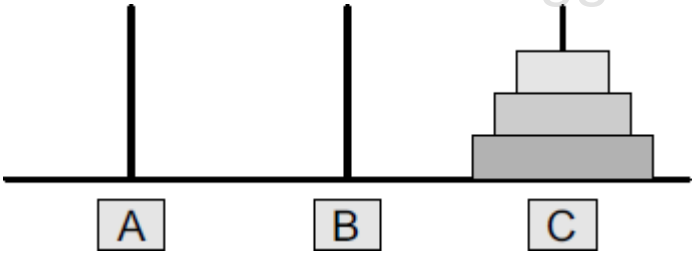


Figure (d)

Example Program

Program for Computation of Sine series

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
```

```
{
   int i, n ;
   float x, val, sum, t ;
   clrscr() ;
   printf("Enter the value for x : ") ;
   scanf("%f", &x) ;
   printf("\nEnter the value for n : ") ;
   scanf("%d", &n) ;
   val = x ;
   x = x * 3.14159 / 180 ;
   t = x ;
   sum = x ;
   for(i = 1 ; i < n + 1 ; i++)
   {
      t = (t * pow((double) (-1), (double) (2 * i - 1)) * x * x) / (2 * i * (2 * i + 1)) ;
      sum = sum + t ;
   }
   printf("\nSine value of %f is : %8.4f", val, sum) ;
   getch() ;
}
```


Output:

```
Enter the value for x : 30
Enter the value for n : 20
Sine value of 30.000000 is : 0.5000
```

Scientific calculator using built-in functions

Program for binary search using recursive function

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

   int a[10],i,n,m,c,l,u;

   printf("Enter the size of an array: ");
   scanf("%d",&n);

   printf("Enter the elements of the array: " );
   for(i=0;i<n;i++)
   {
       scanf("%d",&a[i]);
    }

   printf("Enter the number to be search: ");
   scanf("%d",&m);

   l=0,u=n-1;
   c=binary(a,n,m,l,u);
   if(c==0)
```

```
        printf("Number is not found.");
    else
        printf("Number is found.");

    return 0;
 }

int binary(int a[],int n,int m,int l,int u)
{

    int mid,c=0;

    if(l<=u)
    {
        mid=(l+u)/2;
        if(m==a[mid])
        {
            c=1;
        }
        else if(m<a[mid])
        {
            return binary(a,n,m,l,mid-1);
        }
        else
            return binary(a,n,m,mid+1,u);
    }
    else
```

```
    return c;
}
```

Output:

Enter the size of an array: 5

Enter the elements of the array: 8 9 10 11 12

Enter the number to be search: 8

Number is found.

Pointers

Definition:

A pointer is a variable that stores the address of a variable or a function

Advantages

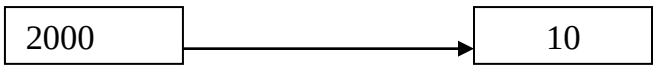
1. Pointers save memory space
2. Faster execution
3. Memory is accessed efficiently.

Declaration

```
datatype    *pointername;
```

E.g) int *p //p is an pointer to an int
float *fp //fp is a pointer to a float

int a=10;
int *p=&a;



p is an integer pointer & holds the address of an int variable a.

Pointer to pointer

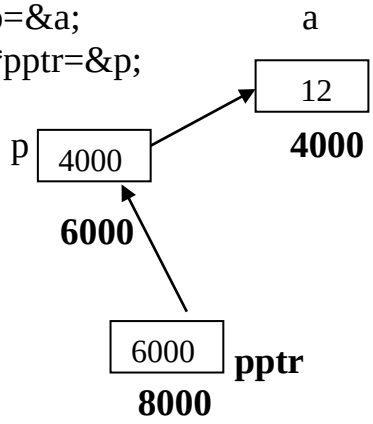
A pointer that holds the address of another pointer variable is known as a pointer to pointer.

E.g.

```
int **p;
```

p is a pointer to a pointer to an integer.

```
int a=12;
int *p=&a;
int **pptr=&p;
```

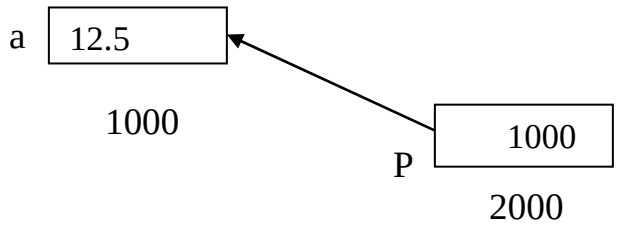


So **pptr=12

Operations on pointers

1. **Referencing operation**: A pointer variable is made to refer to an object. Reference operator(&) is used for this. Reference operator is also known as address of (&) operator.

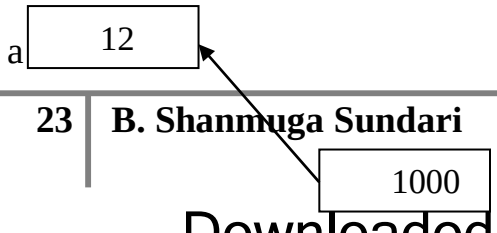
Eg)
```
float a=12.5;
float *p;
p=&a;
```



2. **Dereferencing a pointer**

The object referenced by a pointer can be indirectly accessed by dereferencing the pointer. Dereferencing operator (*) is used for this .This operator is also known as indirection operator or value- at-operator

Eg)




```
int b;
int a=12;
int *p;
```

1000 p=&a;
b=*p; \\value pointed by p(or)value at 1000=12,
so b=12

p
2000

Example program

```
#include<stdio.h>
void main()
{
        int a=12;
        int *p;
        int **pptr;
        p=&a;
        pptr=&p;
        printf("a value=%d",a);
        printf("value by dereferencing p is %d \n",*p);
        printf("value by dereferencing pptr is %d \n",**pptr
        printf("value of p is %u \n",p);
        printf("value of pptr is %u\n",pptr);
}
```

Note

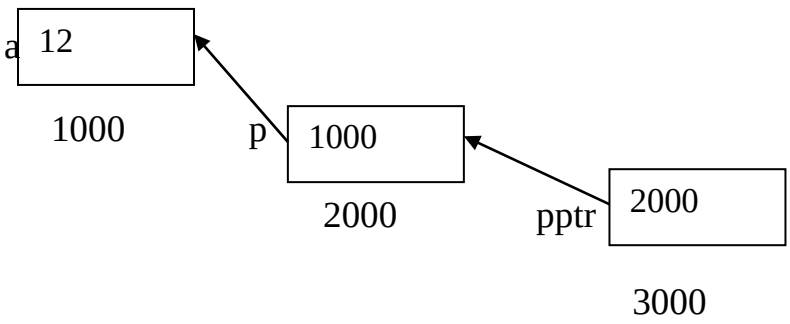
%p is used for addresses; %u can also be used.

*p=value at p
=value at (1000)=12

*pptr=value at(pptr)
=value at(value at (2000))
=value at (1000)=12

Output:

```
a value=12
value by dereferencing p is 12
value by dereferencing pptr is 12
value of p is 1000
value of pptr is 2000
```



Pointer arithmetic

Arithmetic operations on pointer variables are also possible.

E.g.) Addition, Increment, Subtraction, Decrement.

1. Addition

(i) An addition of int type can be added to an expression of pointer type. The result is pointer type.(or)A pointer and an int can be added.

Eg) if p is a pointer to an object then

p+1 =>points to next object

p+i=>point s to ith object after p

(ii)Addition of 2 pointers is not allowed.

2. Increment

Increment operator can be applied to an operand of pointer type.

3. Decrement

Decrement operator can be applied to an operand of pointer type.

4. Subtraction

i) A pointer and an int can be subtracted.

ii) 2 pointers can also be subtracted.

S.no	Operator	Type of operand 1	Type of operand 2	Result type	Example	Initial value	Final value	Description
1	+	Pointer to type T	int	Pointer to type T				Result = initial value of ptr +int operand * sizeof (T)
	Eg.	int *	int	int *	p=p+5	p=2000	2010	2000+5*2= 2010
2	++	Pointer to type T	-	Pointer to type T				**Post increment** Result = initial value of pointer **Pre-increment** Result = initial value of pointer + sizeof (T)
	Eg. post	float*	-	float*	ftr=p++	ftr=?	ftr=2000	

	increment					p=2000	p=2004	Value of ptr = Value of ptr +sizeof(T)
3	-	Pointer to type T	int	Pointer to type T				Result = initial value of ptr - int operand * sizeof (T)
	E.g.	float*	int	float*	p=p-1	p=2000	1996	2000 – 1 * 4 = 2000-4=1996
4	--	Pointer to type T	-	Pointer to type T				**Post decrement** Result = initial value of pointer **Pre-decrement** Result = initial value of pointer – sizeof(T)
	Eg.pre decrement	float*	-	float*	ftr=--p	ftr=? p=2000	ftr=1996 p=1996	Value of ptr

								= Value of ptr – sizeof(T)

3.11. Pointers and Arrays

In C language, pointers and arrays are so closely related.

i) An array name itself is an address or pointer. It points to the address of first element (0^{th} element) of an array.

Example

```
#include<stdio.h>
void main()
{
    int a[3]={10,15,20};
    printf("First element of array is at %u\n", a);
printf("2nd element of array is at %u\n", a+1);
printf("3nd element of array is at %u\n", a+2);
}
```


10	15	20

1000 1002 1004

Output

First element of array is at 1000

2nd element of array is at 1002

3nd element of array is at 1004

ii) Any operation that involves array subscripting is done by using pointers in c language.

E.g.) E1[E2]=>*(E1+E2)

Example

```
#include<stdio.h>
void main()
{
    int a[3]={10,15,20};
    printf("Elements are %d %d %d\n", a[0],a[1],a[2]);
printf("Elements are %d %d %d\n", *(a+0),*(a+1),*(a+2);
}
```

Output:

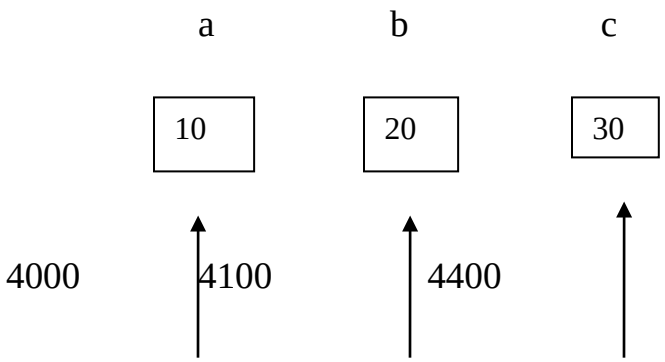
Elements are 10 15 20

Elements are 10 15 20

Array of pointers

An array of pointers is a collection of addresses. Pointers in an array must be the same type.

```
int a=10,b=20,c=30;
int *b[3]={&a,&b,&c};
```



4000	4100	4400

b

5000 5002 5004

Example:

Now look at another code in which we store the address of three individual arrays in the array of pointers:

```
int main()
{
int arr1[]={1,2,3,4,5};
int arr2[]={0,2,4,6,8};
int arr3[]={1,3,5,7,9};
int *parr[3] = {arr1, arr2, arr3};
int i;
for(i = 0;i<3;i++)
printf(«%d», *parr[i]);
return 0;
}
```


Output

1 0 1

Example Program

Sorting of names

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
```

```
char *x[20];
int i,n=0;
void reorder(int n,char *x[]);
clrscr();
printf("Enter no. of String : ");
scanf("%d",&n);
printf("\n");

for(i=0;i<n;i++)
      {
           printf("Enter the Strings %d : ",i+1);
          x[i]=(char *)malloc(20*sizeof(char));
          scanf("%s",x[i]);
      }
          reorder(n,x);
          printf("\nreorder list is : \n");
          for(i=0;i<n;i++)
                {
                  printf("%d %s\n",i+1,x[i]);
                }
                  getch();
}
          void reorder(int n,char *x[])
               {
                  int  i,j;
                  char t[20];
                  for(i=0;i<n-1;i++)
                         for(j=i+1;j<n;j++)
                                if(strcmp(x[i],x[j])>0)
                                  {
                                          strcpy(t,x[j]);
                                          strcpy(x[j],x[i]);
                                          strcpy(x[i],t);
                                  }
                                          return;
                }
```


Output:

Enter no. of string 5

Enter the Strings 1 **kailash**
Enter the Strings 2 **Aswin**
Enter the Strings 3 **Zulphia**
Enter the Strings 4 **Babu**
Enter the Strings 5 **Clinton**
Reorder list is
Aswin
Babu
Clinton
kailash
Clinton

Parameter passing

Whenever we call a function then sequence of executable statements gets executed. We can pass some of the information to the function for processing called **argument.** There are two ways in which arguments can be passed from calling function to called function. They are:

1. Pass by value
2. Pass by reference

1. Pass by value (Call by value)

- In this method the values of actual arguments are copied to formal arguments.
- Any change made in the formal arguments does not affect the actual arguments.
- Once control, return backs to the calling function the formal parameters are destroyed.

E.g. Program:

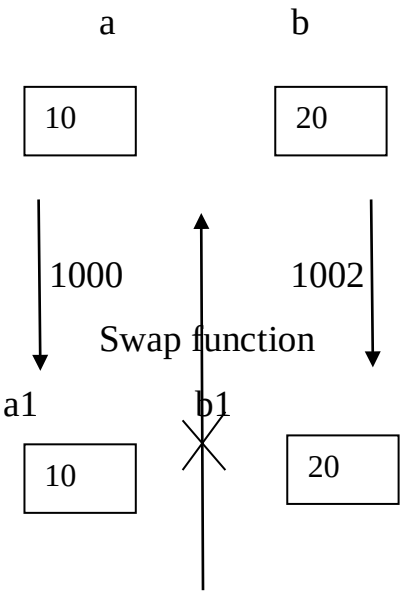
```
#include<stdio.h>
#include<conio.h>
void main()
        {
         int a,b;
         void swap(int ,int);
 a=10;
 b=20;
         printf("\n Before swapping: a = %d  and b = %d",a,b);
         swap(a, b);
 printf("\n After swapping: a= %d and b= %d",a,b);
         getch();
        }

void swap(int a1,int b1)
        {
                int temp;
temp=a1;
a1=b1;
                b1=temp;
        }
```


OUTPUT:

Before swapping: a =10 and b =20

After swapping: a =10 and b = 20

Main function



After swap function

2. Pass by reference (Call b y reference)

- In this method, the addresses of the actual arguments are passed to formal argument.
- Thus formal arguments points to the actual arguments.
- So changes made in the arguments are permanent.

Example Program:

```
#include<stdio.h>
#include<conio.h>
void main()
        {
          int a,b;
          void swap(int *,int *);
          a=10;
  b=20;
          printf("\n Before swapping: a= %d and b= %d",a,b);
          swap(&a,&b);
          printf("\n After swapping: a= %d and b= %d",a,b);
          getch();
        }
void swap(int *a1,int *b1)
        {
        int t;
        t = *a1;
        *a1 = *b1;
        *b1 = t;
        }
```

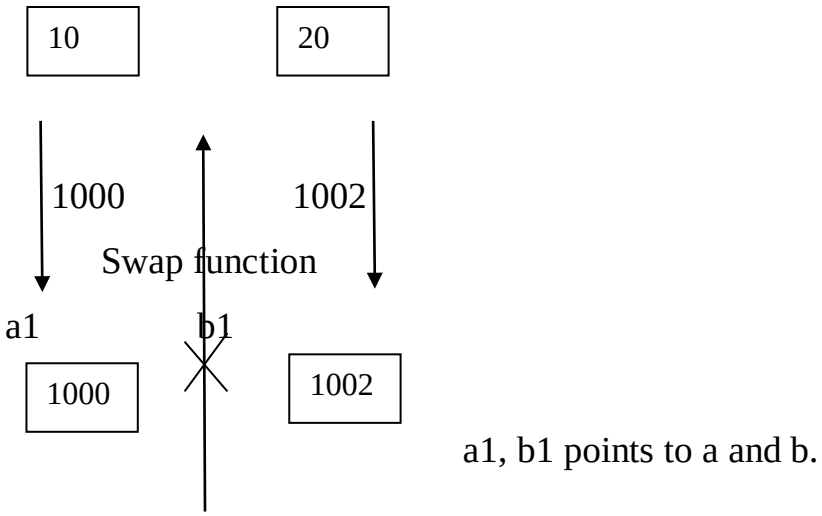

OUTPUT:

Before swapping: a = 10 and b = 20

After swapping: a = 20 and b = 10

Main function

a b



Example Program: Swapping of two numbers and changing the value ofa variable using pass by reference

```
#include<stdio.h>
#include<conio.h>
void swap(int *num1, int *num2);
void main() {
  int x, y;
  printf("\nEnter First number : ");
  scanf("%d", &x);
  printf("\nEnter Second number : ");
  scanf("%d", &y);
```

```
  printf("\nBefore Swaping x = %d and y = %d", x, y);
  swap(&x, &y); // Function Call - Pass By Reference
  printf("\nAfter Swaping x = %d and y = %d", x, y);
  getch();
}
void swap(int *num1, int *num2) {
  int temp;
  temp = *num1;
  *num1 = *num2;
  *num2 = temp;
}
```

Output:


```
Enter First number : 12
Enter Second number : 21

Before Swaping x = 12 and y = 21
After Swaping x = 21 and y = 12
```

UNIT IV STRUCTURES

Structure - Nested structures – Pointer and Structures – Array of structures – Example Program using structures and pointers – Self referential structures – Dynamic memory allocation - Singly linked list - typedef

4.1 Introduction

Using C language we can create new data types. These data types are known as User Defined data types & can be created by using Structures, Unions & Enumerations.

Need for Structure

Arrays can store data of same data type. They can't be used to store data of different data types. For this Structures are used.

Structures

A structure is a collection of variables of different types under a single name. It is used for storing different types of data.

3 aspects:

1. Defining a structure type
2. Declaring variables
3. Using & performing operations.

4.1.1 Structure Definition

The structure can be defined with the keyword **struct** followed by the name of structure and opening brace with data elements of different type then closing brace with semicolon.

General Form

```
struct [structure tag name]
{
type membername1;
type membername2;
……
}[variable name];
```

E.g.

```
struct book
{
    char title[25];
    int pages;
```

float price;

};

- Structure definition does not reserve any space in the memory.
- It is not possible to initialize the structure members during the structure definition.
- A structure definition must always be terminated with a semicolon.

Rules for Structure members

1. A structure can have data of different types
2. A structure can't contain an instance of itself.

E.g.

```
struct box
{
struct box a; // not possible
};
```


3. A structure can contain members of other complete types.

E.g.

```
struct name
{
char firstname[20];
char lastname[20];
};
struct person
{
struct name personname;
float salary;
}
```

4. A structure can contain a pointer to itself;

4.1.2 Declaration

Variables of structure type can be declared either at the time of structure definition or after the structure definition.

General Form

struct structurename variablename[=initialization list];

E.g.

struct book b1,b2;

struct book b3={“CP”,500,385.00};

Accessing Members of Structure

There are two types of operators used for accessing members of a structure.

1. Direct member access operator (dot operator) (.)
2. Indirect member access operator (arrow operator) (->)

Using Dot operator

General form:

structure variable name.member variable name

E.g.

Suppose, we want to access title of structure variable b1, then, it can be accessed as:

b1.title

We can also directly assign values to members.

b1.title= “CP”;

4.2 Structures within a Structure (Nested Structures)

A structure can be nested within another structure. Structure within structure is known as nested structure i.e.) one structure can be declared inside other.

Example program:

```
#include<stdio.h>
struct name
{
    char fname[20],lastname[20];
};
```

```
struct student
{
        int sno,m1,m2,m3;
int tot;
float avg;
struct name sname;
};
void main()
{
struct student s[10];
float,avg;
int n,i;
printf(“Enter the number of students \n”);
scanf(“%d”,&n);
for(i=0;i<n;i++)
{
printf(“Enter student details \n”);
scanf(“%d”,&s[i].sno);
scanf((“%d%d%d”,&s[i].m1, &s[i].m2, &s[i].m3);
scanf(“%s”, s[i].sname.fname);
scanf((“%s”,s[i].sname.lastname);
s[i].tot=s[i].m1+s[i].m2+s[i].m3;
s[i].avg=s[i].tot/6.0;
}
printf(“Student Mark lists\n”);
for(i=0;i<n;i++)
printf(“%s\t%s\t%f”,s[i].sname.fname, s[i].sname.lastname,s[i].avg);
}
```


Output:

Enter the number of students

2

Enter student details

1	70	58	68	AjithKesav
2	90	86	95	RishikKesav

Student Mark lists

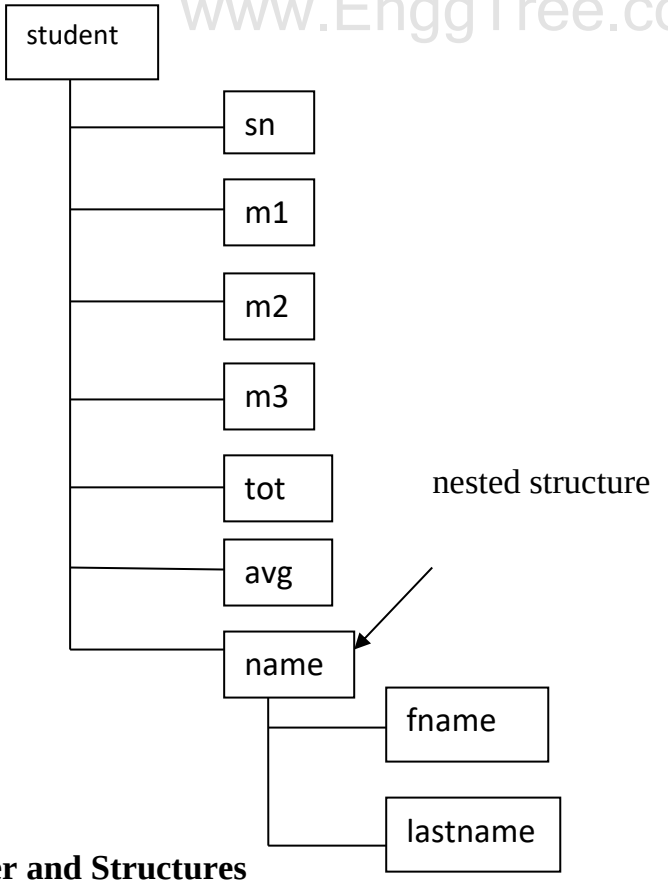
AjithKesav 70.000

RishikKesav 90.000

Accessing members in nested structure

outerstructure variable.innerstructure variable.member name

E.g) s[i].sname.lastname



4.3 Pointer and Structures

It is possible to create a pointer to a structure. A Structure containing a member that is a pointer to the same structure type is called self referential structure. A pointer variable for the structure can be declared by placing an asterisk(*) in front of the structure pointer variable.

Syntax:

struct namedstructuretype
*identifiername;

Example:

```
struct struct_name
{
    data_type member_name1;
    data_type member_name2;
    .....
}*ptr;
OR
struct struct_name *ptr;
```


Dot(.) operator is used to access the data using normal structure variable and arrow (->) is used to access the data using pointer variable.

1. Illustration of Structures using pointers

```
#include <stdio.h>
#include <string.h>
 struct student
{
    int id;
    char name[30];
    float percentage;
};
```

```
int main()
{
    int i;
    struct student record1 = {1, "Raju", 90.5};
    struct student *ptr;
    ptr = &record1;
        printf("Records of Student: \n");
        printf("  Id is: %d \n", ptr->id);
        printf("  Name is: %s \n", ptr->name);
        printf("  Percentage is: %f \n\n", ptr->percentage);
     return 0;
}
```

Output:

Records of Student:

Id is: 1

Name is: Sankar

Percentage is: 90.500000

4.4 Array of Structures

Array of Structures is nothing but a collection of structures. It is an array whose elements are of structure type. This is also called as Structure Array in C.

Consider the structure type struct student. This structure contains student information like student name, s.no etc.

```
struct student
{
        char name[20];
        int sno, m1, m2, m3;
        float average;
};
```

Using a single structure variable we can store single student details. To store information about several students, we have to create a separate variable for each student. It is not feasible. So array of structures are used.

General form

```
struct structurename arrayname[size];
```

E.g. Program:

```
#include<stdio.h>
#include<conio.h>
struct employee
{
        char name[15];
        int empid,bsal;
        float net,gross;
};
```



```
void main()
{
struct employee emp[10];
float hra,da,tax;
int n,i,j;
clrscr();
printf("Enter the number of employees\n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
 printf("\nEnter the employee name");
 scanf("%s",emp[i].name);
 printf("\nEnter the employee id");
```

```
 scanf("%d",&emp[i].empid);
 printf("\nEnter the basic salary");
 scanf("%d",&emp[i].bsal);
 hra=((10*emp[i].bsal)/100);
 da=((35*emp[i].bsal)/100);
 tax=((15*emp[i].bsal)/100);
 emp[i].gross=emp[i].bsal+hra+da;
 emp[i].net=emp[i].gross-tax;
}
printf("Employee Name Employee ID Employee Net Salary \n");
for(i=1;i<=n;i++)
printf("%s\t\t%d\t\t%f\n",emp[i].name,emp[i].empid,emp[i].net);
getch();
}
```


Output:

```
Enter the number of Employees
2
Enter the employee name
Anu
Enter the employee id
01
Enter the basic salary
1000
Enter the employee name
Meena
Enter the employee id
02
Enter the basic salary
```

2000

Employee Name	Employee ID	Net Salary
Anu	01	1300.000
Meena	02	2600.000

4.5 Example Program using structures and pointers

1. C program to read and print employee's record using structure

```
#include <stdio.h>
 /*structure declaration*/
struct employee{
    char    name[30];
    int     empId;
    float   salary;
};

int main()
{
    /*declare structure variable*/
    struct employee emp;

    /*read employee details*/
    printf("\nEnter details :\n");
    printf("Name ?:");          gets(emp.name);
    printf("ID ?:");           scanf("%d",&emp.empId);
    printf("Salary ?:");        scanf("%f",&emp.salary);

    /*print employee details*/
    printf("\nEntered detail is:");
    printf("Name: %s"   ,emp.name);
```

```
    printf("Id: %d"     ,emp.empId);
    printf("Salary: %f\n",emp.salary);
    return 0;
}
```

Output:

Enter details :

Name ?:Raju

ID ?:007

Salary ?:76543

Entered detail is:

Name: Raju

Id: 007

Salary: 76543.000000

4.6 Self referential structures

Self referential Structures are those structures that contain a reference to data of its same type. i.e in addition to other data a self referential structure contains a pointer to a data that it of the same type as that of the structure. For example: consider the structure node given as follows:

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

Here the structure node will contain two types of data an integer val and next which is a pointer a node. Self referential structure is the foundation of other data structures.

4.7 Dynamic memory allocation

Dynamic memory allocation refers to the process of manual memory management (allocation and deallocation).

The functions supports for dynamic memory allocation are,

1. malloc()
2. calloc()
3. realloc()
4. free()

1. malloc() function

malloc() allocates N bytes in memory and return pointer to allocated memory. The returned pointer contains link/handle to the allocated memory.

void * malloc(number_of_bytes);

- It returns void pointer (generic pointer). Which means we can easily typecast it to any other pointer types.
- It accepts an integer number_of_bytes, i.e. total bytes to allocate in memory.

Note: malloc() returns NULL pointer on failure.

Example

```
int N = 10; // Number of bytes to allocate
int *ptr;   // Pointer variable to store address
ptr = (int *) malloc(N * sizeof(int));  // Allocate 10 * 4 bytes in memory
```

Here,

- ptr is a pointer to integer to store address of the allocated memory.
- (int *) is typecast required. As, I mentioned above that malloc() return void *. Hence, to work with void pointer we must typecast it to suitable type.
- N * sizeof(int) - Since size of int is not fixed on all compilers. Hence, to get size of integer on current compiler I have used sizeof() operator.

2. **calloc() function**

calloc() function allocates memory contiguously. It allocates multiple memory blocks and initializes all blocks with 0 (NULL).

Note: malloc() allocates uninitialized memory blocks.

Syntax

void* calloc(number_of_blocks, number_of_bytes);

Here,

- Similar to malloc() it returns void pointer.
- It accepts two parameters number_of_blocks i.e. total blocks to allocate and number_of_bytes i.e. bytes to allocate per block.

Therefore, you can say that calloc() will allocate

total (number_of_blocks * number_of_bytes) bytes. Each block initialized with 0 (NULL).

Example:

int *ptr;

ptr = (int *) calloc(N, sizeof(int));

Here, all memory blocks are initialized with 0.

3. realloc() function

When working with huge data and if the allocated memory is not sufficient to store data. In that case, we need to alter/update the size of an existing allocated memory blocks (which has been created by either malloc() or calloc()).

We use realloc() function to alter/update the size of exiting allocated memory blocks. The function may resize or move the allocated memory blocks to a new location.

Syntax

void* realloc(ptr, updated_memory_size);

- Similar to all other functions for Dynamic Memory Allocation in C, it returns void pointer. Which points to the address of existing or newly allocated memory.
- ptr is a pointer to memory block of previously allocated memory.
- updated_memory_size is new (existing + new) size of the memory block.

Example

// Original memory blocks allocation

```
int N = 10;
int *ptr;
ptr = (int *) malloc(N * sizeof(int));

// Increase the value of N
N = 50;
// Reallocate memory blocks
ptr = (int *) realloc(ptr, N * sizeof(int));
```

4. **free() function**

C programming has a built-in library function free() to clear or release the unused memory.

The free() function clears the pointer (assigns NULL to the pointer) to clear the dynamically allocated memory. If pointer contains NULL, then free() does nothing (because pointer will not be pointing at any memory addresses). If it contains any address of dynamically allocated memory, free() will clear pointer by assigning NULL.

Syntax

```
free(ptr);
```

The function accepts a void pointer ptr. It points to previously allocated memory using any of Dynamic Memory Allocation functions in C.

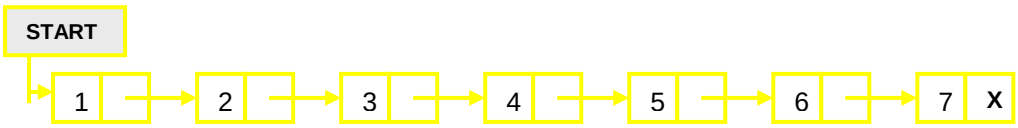
Example:

```
int N=10;
int *ptr;
// Allocate memory using malloc
ptr=(int *) malloc (N* size of (int));
//Free allocated memory
free(ptr);
```

4.8 Singly Linked List

- A linked list in simple terms is a linear collection of data elements. These data elements are called nodes.

- Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as building block to implement data structures like stacks, queues and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contain one or more data fields and a pointer to the next node.



In the above linked list, every node contains two parts- one integer and the other a pointer to the next node. The left part of the node which contains data may include a simple data type, an array or a structure. The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL.

A **singly linked list** is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node we mean that the node stores the address of the next node in sequence.

4.8.1 Traversing a singly linked list

Traversing a linked list means accessing the nodes of the list in order to perform some operations on them. A Linked list always contains a pointer variable START which stores the address of the first node of the list. The end of the list is marked by string NULL or -1 in the NEXT field of the last node. For traversing the singly linked list, we make use of another pointer variable PTR which points to the node that is correctly being accessed. The algorithm to traverse a linked list is shown below:

```
Algorithm for traversing a linked list
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:         Apply Process to PTR->DATA
Step 4:         SET PTR = PTR->NEXT
     [END OF LOOP]
Step 5: EXIT
```

In this algorithm, we first initialize PTR with the address of start. So now PTR points to the first node of the linked list.

Then in step 2 while loop is executed which is repeated till PTR processes the last node, that is, until it encounters NULL.

In step 3, we apply the process to the current node.

In step 4, we move to the next node by making PTR point to the node whose address is stored in the NEXT field.

The algorithm print the information stored in each node of the linked list is shown below:

```
Algorithm to print the information stored in
each node of the linked list
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:         Write PTR->DATA
Step 4:         SET PTR = PTR->NEXT
      [END OF LOOP]
Step 5: EXIT
```

We will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is when all the nodes of the linked list have been traversed, the final value of the counter will be displayed. Figure below shows the algorithm to print the number of nodes in a linked list.

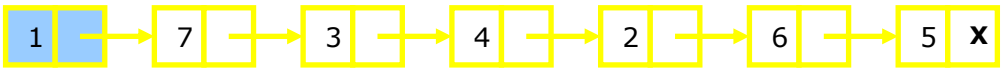
```
Algorithm to print the number of nodes in the linked list
Step 1: [INITIALIZE] SET Count = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:           SET Count = Count + 1
Step 5:           SET PTR = PTR->NEXT
      [END OF LOOP]
```

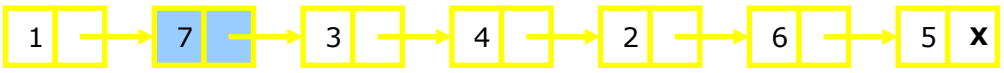
4.8.2 Searching for a value in a Linked list

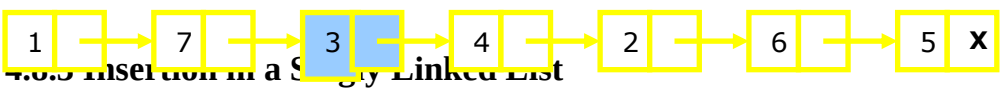
Searching a linked list means to find a particular element in the linked list. A linked list consists of two parts – the DATA part and NEXT part, where DATA stores the relevant information and NEXT stores the address of the next node in the sequence. Figure below shows the algorithm to search a linked list.

```
Algorithm to search an unsorted linked list
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 while PTR != NULL
Step 3:           IF VAL = PTR->DATA
                  SET POS = PTR
                  Go To Step 5
            ELSE
                  SET PTR = PTR->NEXT
            [END OF IF]
      [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```


Consider the linked list shown in figure we have val=4, then the flow of the algorithm can be explained as shown in figure







4.8.3 Insertion in a Singly Linked List

Insert a new node at the head of the list is straightforward. The main idea is that we create a new node, set its next link to refer to the current head, and then set head to point to the new node.

Algorithm addFirst(String newData):

```
create a new node v containing newData
v.setNext(head)
head = v
size = size + 1
```

4.8.3 Insertion at the tail

If we keep a reference to the tail node, then it would be easy to insert an element at the tail of the list. Assume we keep a tail node in the class of SLinkedList, the idea is to create a new node, assign its next reference to point to a null object, set the next reference of the tail to point to this new object, and then assign the tail reference itself to this new node. Initially both head and tail point to null object.

Algorithm addLast(String newData):


```
create a new node v containing newData
v.setNext(null)
if (head == null) { // list is empty
    head = v
} else { // list is not empty
    tail.setNext(v)
}
tail = v
size = size + 1
```

4.8.4 Deletion in a Singly Linked List

Deletion at the head

Removal of an element at the head of a singly linked list is relatively easy. However removing a tail node is not easy.

Algorithm removeFirst()

if (head = = null) then

Indicate an error: the list is empty

tmp = head

head = head.getNext()

tmp.setNext(null)

size = size - 1

Example:

```
include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
struct test_struct
{
int val;
struct test_struct *next;
};
struct test_struct *head = NULL;
struct test_struct *curr = NULL;
struct test_struct* create_list(int val)
{
printf("\n creating list with headnode as [%d]\n",val);
struct test_struct *ptr = (struct test_struct*)malloc(sizeof(struct test_struct));
if(NULL == ptr)
{
printf("\n Node creation failed \n"); return NULL;
}
ptr->val = val;
ptr->next = NULL;
head = curr = ptr;
return ptr;
```

```
}
struct test_struct* add_to_list(int val, bool add_to_end)
{
if(NULL == head)
{
return (create_list(val));
}
if(add_to_end)
printf("\n Adding node to end of list with value [%d]\n",val);
else printf("\n Adding node to beginning of list with value [%d]\n",val);
struct test_struct *ptr = (struct test_struct*)malloc(sizeof(struct test_struct));
if(NULL == ptr)
{ printf("\n Node creation failed \n"); return NULL;
}
ptr->val = val;
ptr->next = NULL;
if(add_to_end)
{ curr->next = ptr; curr = ptr;
}
else { ptr->next = head; head = ptr;
}
return ptr;
}
struct test_struct* search_in_list(int val, struct test_struct **prev)
{
struct test_struct *ptr = head;
struct test_struct *tmp = NULL;
bool found = false;
printf("\n Searching the list for value [%d] \n",val);
while(ptr != NULL)
{
```

```
if(ptr->val == val)
{ found = true; break;
}
else
{
tmp = ptr; ptr = ptr->next;
}
}
if(true == found)
{
if(prev) *prev = tmp;
return ptr;
}
else
{
return NULL; } }
int delete_from_list(int val)
{
struct test_struct *prev = NULL;
struct test_struct *del = NULL;
printf("\n Deleting value [%d] from list\n",val);
del = search_in_list(val,&prev);
if(del == NULL) { return -1; }
else
{
if(prev != NULL)
prev->next = del->next;
if(del == curr)
{ curr = prev;
}
else if(del == head)
```

```
{
head = del->next;
} }
free(del);
del = NULL; return 0;
}
void print_list(void) {
struct test_struct *ptr = head;
printf("\n -------Printing list Start------- \n");
while(ptr != NULL)
{
printf("\n [%d] \n",ptr->val);
ptr = ptr->next;
}
printf("\n -------Printing list End------- \n"); return;
}
int main(void)
{
int i = 0, ret = 0;
struct test_struct *ptr = NULL;
print_list();
for(i = 5; i<10; i++)
add_to_list(i,true); print_list();
for(i = 4; i>0; i--)
add_to_list(i,false);
print_list();
for(i = 1; i<10; i += 4)
{ ptr = search_in_list(i, NULL);
if(NULL == ptr)
{
printf("\n Search [val = %d] failed, no such element found\n",i);
```

```
}
else
{
printf("\n Search passed [val = %d]\n",ptr->val); }
print_list(); ret = delete_from_list(i);
if(ret != 0) { printf("\n delete [val = %d] failed, no such element found\n",i);
}
else
{
printf("\n delete [val = %d] passed \n",i); } print_list(); } return 0;
}
```

4.9 Typedef

The typedef keyword enables the programmer to create a new data type name by using an existing data type.

By using typedef, no new data is created, rather an alternate name is given to a known data type.

Syntax: typedef existing_data_type new_data_type;

It is used to create a new data using the existing type.

Syntax: typedef data type name;

Example: typedef int hours: hours hrs;/

* Now, hours can be used as new datatype */

Multiple Choice Questions

1. **A data structure that can store related information together is**
   A. array
   B. string
   C. structure
   D. all of these

Answer: all of these

2. **A data structure that can store related information of different data types together is**

A. array
B. string
C. Structure
D. all of these

Answer: Structure

3. **Memory for a structure is allocated at the time of**

A. Structure definition
B. Structure variable declaration
C. Structure declaration
D. Function declaration

Answer: Structure variable declaration

4. **A Structure member variable is generally accessed using the**

A. address operator
B. dot operator
C. comma operator
D. ternary operator

Answer: dot operator

5. **A Structure can be placed within another structure and is known as,**

A. Self-referential structure
B. Nested Structure
C. Parallel structure
D. Pointer to structure

Answer: Nested Structure

6. **A union number variable is generally accessed using the**

A. address operator

B. dot operator

C. comma operator

D. ternary operator

Answer: dot operator

7. **Typedef can be used with which of these data types**

A. struct

B. union

C. enum

D. all of these

Answer: all of these

8. **The enumerated type is derived from which data type**

A. int

B. double

C. float

D. char

Answer: int

9. **Which operator connects the structure name to its member name?**

A. –

B. <-

C. Dot operator

D. Both <- and dot operator

Answer: Dot operator

10. **Which of the following operation is illegal in structures?**

a) Typecasting of structure

b) Pointer to a variable of the same structure

c) Dynamic allocation of memory for structure

d) All of the mentioned

Answer: Typecasting of structure

11. **Presence of code like "s.t.b = 10" indicates _____**

a) Syntax Error

b) Structure

c) double data type

d) An ordinary variable name

Answer: Structure

12. **What is the size of a C structure.?**

A) C structure is always 128 bytes.

B) Size of C structure is the total bytes of all elements of structure.

C) Size of C structure is the size of largest element.

D) None of the above

Answer: Size of C structure is the total bytes of all elements of structure

13. **A C Structure or User defined data type is also called.?**

A) Derived data type

B) Secondary data type

C) Aggregate data type

D) All the above

Answer: All the above

14. **What is actually passed if you pass a structure variable to a function.?**

A) Copy of structure variable

B) Reference of structure variable

C) Starting address of structure variable

D) Ending address of structure variable

Answer: Copy of structure variable

15. **What is the output of C program with structure array pointers.?**

```
int main()
{
   struct car
   {
      int km;
   }*p1[2];
   struct car c1={1234};
   p1[0]=&c1;
   printf("%d ",p1[0]->km);
   return 0;
}
```

A) 0

B) 1

C) c

D) Compiler error

Answer: 1234

UNIT V FILE PROCESSING

Files – Types of file processing: Sequential access, Random access – Sequential access file - Example Program: Finding average of numbers stored in sequential access file - Random access file -Example Program: Transaction processing using random access files – Command line arguments

5.1 Introduction

A file is a semi-permanent, named collection of data. A File is usually stored on magnetic media, such as a hard disk or magnetic tape. Semi-permanent means that data saved in files stays safe until it is deleted or modified.

Named means that a particular collection of data on a disk has a name, like mydata.dat and access to the collection is done by using its name.

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure.

5.1.1 Types of Files

1. Text files
2. Binary files

1. Text Files

A text file consists of consecutive characters, which are interpreted by the library functions used to access them and by format specifiers used in functions.

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

2. Binary files

A binary file consists of bytes of data arranged in continuous block. A separate set of library functions is there to process such data files.

Binary files are mostly the .bin files in your computer. Instead of storing data in plain text, they store it in the binary form (0's and 1's). They can hold higher amount of data, are not readable easily and provides a better security than text files.

5.1.2 File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file

2. Opening an existing file

3. Closing a file

4. Reading from and writing information to a file

Function description

fopen() - create a new file or open a existing file

fclose() - closes a file

getc() - reads a character from a file

putc() - writes a character to a file

fscanf() - reads a set of data from a file

fprintf() - writes a set of data to a file

getw() - reads a integer from a file

putw() - writes a integer to a file

fseek() - set the position to desire point

ftell() - gives current position in the file

rewind() - sets the position to the beginning point

1. Opening a File or Creating a File

Opening a file means creating a new file with specified file name and with accessing mode.

The fopen() function is used to create a new file or to open an existing file.

Syntax:

****fp = FILE *fopen(const char *filename, const char *mode);***

Here, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file.

filename is the name of the file to be opened and mode specifies the purpose of opening the file.

Mode can be of following types:

Mode Description

Mode	Purpose
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

2. Closing a File

A file must be closed after all the operation of the file have been completed. The fclose() function is used to close an already opened file.

Syntax:

int fclose(FILE *fp);

Here fclose() function closes the file and returns zero on success, or EOF if there is an error in closing the file. This EOF is a constant defined in the header file stdio.h.

3. Reading and writing to a text file

i. Read a character from a file: fgetc function

The 'fgetc' function is used to read a character from a file which is opened in read mode.

Syntax:

c=fgetc(p1);

where p1 is the file pointer.

ii. Read a data from a file: fscanf function

The fscanf function is used to read data from a file. It is similar to the scanf function except that fscanf() is used to read data from the disk.

Syntax:

fscanf(fb "format string", &v1, &v2…&vn);

where fb refers to the file pointer. v1, v2, … vn refers variables whose values are read from the disk "format string" refers the control string which represents the conversion specification.

iii. Write a character to a file:fputc function

The function 'fputc' is used to write a character variable x to the file opened in write mode.

Syntax:

fputc(x,fp1);

where fp1 is the file pointer.

iv. Writing data to a file : fprintf()

fprintf() function is used to write data to a file. It is similar to the printf() function except that fprintf() is used to write data to the disk.

Syntax:

fprintf(fp, "format string", v1,v2… vn);

where fp refers to the file pointer.

5.2 Types of file processing

There are two main ways a file can be organized:

1. **Sequential Access** — In this type of file, the data are kept sequentially. To read last record of the file, it is expected to read all the records before that particular record. It takes more time for accessing the records.

2. **Random Access** — In this type of file, the data can be read and modified randomly. If it is desired to read the last record of a file, directly the same record can be read. Due to random access of data, it takes less access time as compared to the sequential file.

Sequential Access File

A Sequential file is characterized by the fact that individual data items are arranged serially in a sequence, one after another. They can only be processed in serial order from the beginning. In other words, the records can be accessed in the same manner in which they have been stored. It is not possible to start reading or writing a sequential file from anywhere except at the beginning.

Random Access File

The second and better method of arranging records of a file is called direct access or random access. In this arrangement one can have access to any record which is situated at the middle of the file without reading or passing through other records in the file.

5.3 Reading Sequential Access file

Data is stored in files so that the data can be retrieved for processing when needed

Example:

clients.dat file contents

100 Jones 9023.00

200 Frank 234.00

300 Mano 29023.00

400 Bala 2344.00

Program:

```
// Reading and printing a sequential file

#include <stdio.h>
```

```
#include <stdlib.h>
int main(void) {
  unsigned int account;  // account number
  char name[30];         // account name
  double balance;        // account balance
  FILE *cfPtr;  // cfPtr = clients.dat file pointer
  // fopen opens file; exits program if file cannot be opened
  if ((cfPtr = fopen("clients.dat", "r")) == NULL) {
    puts("File could not be opened");
exit(0);
  }
   printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
  fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
  // while not end of file
  while (!feof(cfPtr)) {
    printf("%-10d%-13s%7.2f\n", account, name, balance);
    fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
  }  // end while

  fclose(cfPtr);  // fclose closes the file
}  // end main
```


Output:

```
Account   Name         Balance
100       Jones        9023.00
200       Frank        234.00
300       Mano         29023.00
```

400 Bala 2344.00

5.4 Read numbers from file and calculate Average

```
/* Program to read from the num.dat file  and find the average of the numbers  */
#include <stdio.h>
#include <stdlib.h>
#define DATAFILE "prog15.dat"
 int main() {
  FILE* fp;
  int n[50], i = 0;
  float sum = 0;
  if ((fp = fopen(DATAFILE, "r")) == NULL) {
    printf("Unable to open %s...\n", DATAFILE);
    exit(0);
  }
  puts("Reading numbers from num.dat");
  while (!feof(fp)) {
    fscanf(fp, "%d ", &n[i]);
    printf("%d %d\n", i, n[i]);
    sum += n[i];
    i++;
  }
  fclose(fp);
  // if no data is available in the file
  if (i == 0)
    printf("No data available in %s", DATAFILE);
```

```
 float average = sum / i;
 printf("The average is %.3f for %d numbers\n", average, i);
return 0;
}
```

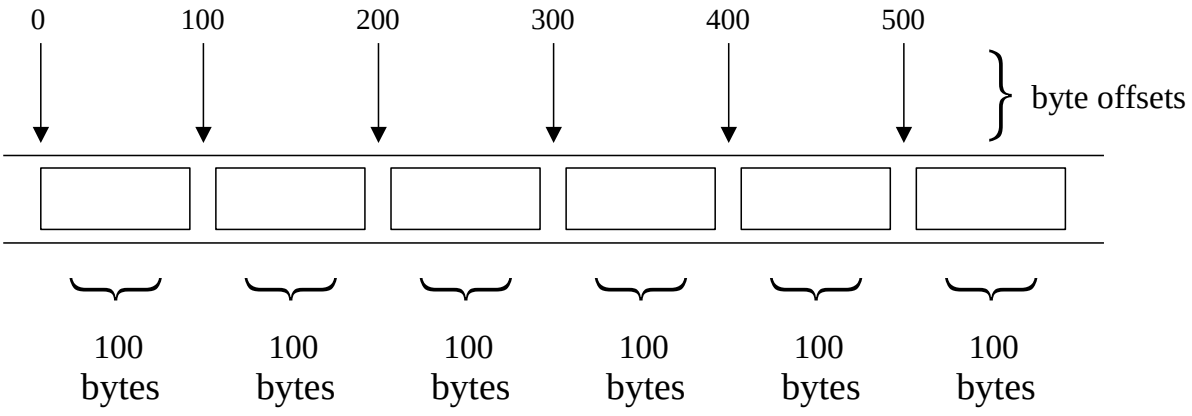
Output:

```
> prog15
Reading numbers from prog15.dat
0 90
1 10
2 20
3 30
4 40
5 50
The average is 40.000 for 6 numbers
>
```

5.5 Random access file

- Access individual records without searching through other records
- Instant access to records in a file
- Data can be inserted without destroying other data
- Data previously stored can be updated or deleted without overwriting.

- Implemented using fixed length records
  - Sequential files do not have fixed length records



5.5.1 Functions For Selecting A Record Randomly

The functions used to randomly access a record stored in a file are fseek(), ftell(), rewind(), fgetpos(), and fsetpos().

1. fseek()

- fseek() is used to reposition a binary stream. The prototype of fseek() can be given as,
- int fseek(FILE *stream, long offset, int origin);
- fseek() is used to set the file position pointer for the given stream. Offset is an integer value that gives the number of bytes to move forward or backward in the file. Offset may be positive or negative, provided it makes sense. For example, you cannot specify a negative offset if you are starting at the beginning of the file. The *origin* value should have one of the following values (defined in stdio.h):
- SEEK_SET: to perform input or output on offset bytes from start of the file
- SEEK_CUR: to perform input or output on offset bytes from the current position in the file
- SEEK_END: to perform input or output on offset bytes from the end of the file
- SEEK_SET, SEEK_CUR and SEEK_END are defined constants with value 0, 1 and 2 respectively.
- On successful operation, fseek() returns zero and in case of failure, it returns a non-zero value. For example, if you try to perform a seek operation on a file that is not opened in binary mode then a non-zero value will be returned.
- fseek() can be used to move the file pointer beyond a file, but not before the beginning.

Example: Write a program to print the records in reverse order. The file must be opened in binary mode. Use fseek()

```
#include<stdio.h>

#include<conio.h>

main()

{       typedef struct employee
```

```
{       int emp_code;
        char name[20];
        int hra;
        int da;
        int ta;
};
FILE *fp;
struct employee e;
int result, i;
fp =  fopen("employee.txt", "rb");
if(fp==NULL)
{       printf("\n Error opening file");
        exit(1);
}
for(i=5;i>=0;i--)
{       fseek(fp, i*sizeof(e), SEEK_SET);
        fread(&e, sizeof(e), 1, fp);
        printf("\n  EMPLOYEE CODE : %d", e.emp_code);
        printf("\n Name : %s", e.name);
        printf("\n HRA, TA and DA : %d %d %d", e.hra, e.ta, e.da);
}
fclose(fp);
getch();
return 0;
}
```


2. rewind()

- rewind() is used to adjust the position of file pointer so that the next I/O operation will take place at the beginning of the file. It's prototype can be given as
- **void rewind(FILE *f);**
- rewind() is equivalent to calling fseek() with following parameters: fseek(f,0L,SEEK_SET);

3. fgetpos()

- The fgetpos() is used to determine the current position of the stream. It's prototype can be given as

  int fgetpos(FILE *stream, fpos_t *pos);

- Here, stream is the file whose current file pointer position has to be determined. pos is used to point to the location where fgetpos() can store the position information. The pos variable is of type fops_t which is defined in stdio.h and is basically an object that can hold every possible position in a FILE.
- On success, fgetpos() returns zero and in case of error a non-zero value is returned. Note that the value of pos obtained through fgetpos() can be used by the fsetpos() to return to this same position.

4. fsetpos()

- The fsetpos() is used to move the file position indicator of a stream to the location indicated by the information obtained in "pos" by making a call to the fgetpos(). Its prototype is
- int fsetpos(FILE *stream, const fops_t pos);
- Here, stream points to the file whose file pointer indicator has to be re-positioned. pos points to positioning information as returned by "fgetpos".
- On success, fsetpos() returns a zero and clears the end-of-file indicator. In case of failure it returns a non-zero value

The program opens a file and reads bytes at several different locations.

```
#include <stdio.h>

main()

{
```

```
FILE   *fp;
fpos_t pos;
char   feedback[20];
fp = fopen(“comments.txt”, “rb”);
if(fp == NULL)
{
        printf(“\n Error opening file”);
        exit(1);
}
// Read some data and then check the position.
fread( feedback, sizeof(char), 20, fp);
if( fgetpos(fp, &pos) != 0 )
{
        printf(“\n Error in fgetpos()");
        exit(1);
}
fread(feeback, sizeof(char), 20, fp);
printf("\n 20 bytes at byte %ld: %s", pos, feedback);
        // Set a new position and read more data
pos = 90;
if( fsetpos(fp, &pos ) != 0 )
{
        printf(“\n Error in fsetpos()");
        exit(1);
}
fread( feedback, sizeof(char), 20, fp);
```

```
        printf( "\n 20 bytes at byte %ld: %s", pos, feedback);
        fclose(fp);
}
```

5. ftell()

The ftell function is used to know the current position of file pointer. It is at this position at which the next I/O will be performed. The syntax of the ftell() defined in stdio.h can be given as:

long ftell (FILE *stream);

On successful, ftell() function returns the current file position (in bytes) for stream. However, in case of error, ftell() returns -1.

When using ftell(), error can occur either because of two reasons:

First, using ftell() with a device that cannot store data (for example, keyboard)

Second, when the position is larger than that can be represented in a long integer. This will usually happen when dealing with very large files

```
FILE *fp;
char c;
int n;
fp=fopen("abc","w");
if(fp==NULL)
{       printf("\n Error Opening The File");
        exit(1);
}
while((c=getchar())!=EOF)
putc(c,fp);
n = ftell(fp);
fclose(fp);
fp=fopen("abc","r");
```

```
if(fp==NULL)
{       printf("\n Error Opening The File");
        exit(1);
}
while(ftell(fp)<n)
{       c= fgetc(fp);
        printf('%c", c);
}
fclose(fp);
```

5.6 Example Program: Transaction processing using random access files

The program maintains a bank's account information—updating existing accounts, adding new accounts, deleting accounts and storing a listing of all the current accounts in a text file for printing.

The program has five options.

Option 1

calls function textFile to store a formatted list of all the accounts (typically called a report) in a text file called accounts.txt that may be printed later. The function uses fread and the sequential file access techniques used in the program of Section below.

After **option 1** is chosen, the file **accounts.txt** contains:

Acct	*Last Name*	*First Name*	*Balance*
29	*Brown*	*Nancy*	*-24.54*
33	*Dunn*	*Stacey*	*314.33*
37	*Barker*	*Doug*	*0.00*
88	*Smith*	*Dave*	*258.34*
96	*Stone*	*Sam*	*34.98*

Option 2

calls the function **updateRecord** to update an account. The function will update only a record that already exists, so the function first checks whether the record specified by the user is empty. The record is read into structure client with fread, then member acctNum is compared to 0. If it's 0, the record contains no information, and a message is printed stating that the record is empty. Then the menu choices are displayed. If the record contains information, function updateRecord inputs the transaction amount, calculates the new balance and rewrites the record to the file.

Enter account to update (1 - 100): 37

37 Barker Doug 0.00

Enter charge (+) or payment (-): +87.99

37 Barker Doug 87.99

Option 3

calls the function **newRecord** to add a new account to the file. If the user enters an account number for an existing account, newRecord displays an error message indicating that the record already contains information, and the menu choices are printed again

Enter new account number (1 - 100): 22

Enter lastname, firstname, balance

? Johnston Sarah 247.45

Option 4

calls function deleteRecord to delete a record from the file. Deletion is accomplished by asking the user for the account number and re-initialising the record. If the account contains no information, deleteRecord displays an error message indicating that the account does not exist.

Option 5

terminates program execution.

Example Program:

```
// Bank-account program reads a random-access file sequentially, updates data already written to the file, creates new data to be placed in the file, and deletes data previously in the file. //

#include <stdio.h>

#include <stdlib.h>

// clientData structure definition

struct clientData {

  unsigned int acctNum; // account number

  char lastName[15];    // account last name

  char firstName[10];   // account first name

  double balance;       // account balance

};                      // end structure clientData

 // prototypes

unsigned int enterChoice(void);

void textFile(FILE *readPtr);

void updateRecord(FILE *fPtr);

void newRecord(FILE *fPtr);

void deleteRecord(FILE *fPtr);

int main(int argc, char *argv[]) {

  FILE *cfPtr;         // credit.dat file pointer

  unsigned int choice; // user's choice

  // fopen opens the file; exits if file cannot be opened

  // Do not change the mode "rb+" - it will not work!

  if ((cfPtr = fopen("credit.dat", "rb+")) == NULL) {

    printf("%s: File could not be opened.\n", argv[0]);

    exit(-1);
```

```
  }
  // enable user to specify action
  while ((choice = enterChoice()) != 5) {
    switch (choice) {
    // create text file from record file
    case 1:
      textFile(cfPtr); break;
    // update record
    case 2:
      updateRecord(cfPtr); break;
    // create record
    case 3:
      newRecord(cfPtr); break;
    // delete existing record
    case 4:
      deleteRecord(cfPtr); break;
    // display if user does not select valid choice
    default:
      puts("Incorrect choice"); break;
    } // end switch
  } // end while
  fclose(cfPtr); // fclose closes the file
} // end main
// create formatted text file for printing
void textFile(FILE *readPtr) {
  FILE *writePtr; // accounts.txt file pointer
```

```
  int result;     // used to test whether fread read any bytes
  // create clientData with default information
  struct clientData client = {0, "", "", 0.0};
  // fopen opens the file; exits if file cannot be opened
  if ((writePtr = fopen("accounts.txt", "w")) == NULL) {
    puts("File could not be opened.");
  } // end if
  else {
    rewind(readPtr); // sets pointer to beginning of file
    fprintf(writePtr, "%-6s%-16s%-11s%10s\n", "Acct", "Last Name","First Name",
"Balance");
    // copy all records from random-access file into text file
    while (!feof(readPtr)) {
      result = fread(&client, sizeof(struct clientData), 1, readPtr);
      // write single record to text file
      if (result != 0 && client.acctNum != 0) {
        fprintf(writePtr, "%-6d%-16s%-11s%10.2f\n", client.acctNum,
                client.lastName, client.firstName, client.balance);
      } // end if
    }   // end while

    fclose(writePtr); // fclose closes the file
  }                 // end else
} // end function textFile
 // update balance in record
void updateRecord(FILE *fPtr) {
```

```
  unsigned int account; // account number
  double transaction;   // transaction amount
  // create clientData with no information
  struct clientData client = {0, "", "", 0.0};
   // obtain number of account to update
  printf("%s", "Enter account to update ( 1 - 100 ): ");
  scanf("%d", &account);
   // move file pointer to correct record in file
  fseek(fPtr, (account - 1) * sizeof(struct clientData), SEEK_SET);
  // read record from file
  fread(&client, sizeof(struct clientData), 1, fPtr);
  // display error if account does not exist
  if (client.acctNum == 0) {
    printf("Account #%d has no information.\n", account);
  } else { // update record
    printf("%-6d%-16s%-11s%10.2f\n\n", client.acctNum, client.lastName,
          client.firstName, client.balance);
    // request transaction amount from user
    printf("%s", "Enter charge ( + ) or payment ( - ): ");
    scanf("%lf", &transaction);
    client.balance += transaction; // update record balance
printf("%-6d%-16s%-11s%10.2f\n", client.acctNum, client.lastName,
client.firstName, client.balance);
// move file pointer to correct record in file
// move back by 1 record length
fseek(fPtr, -sizeof(struct clientData), SEEK_CUR);
```

```
 // write updated record over old record in file
    fwrite(&client, sizeof(struct clientData), 1, fPtr);
  } // end else
} // end function updateRecord

// delete an existing record
void deleteRecord(FILE *fPtr) {
  struct clientData client; // stores record read from file
  struct clientData blankClient = {0, "", "", 0}; // blank client
  unsigned int accountNum;                    // account number

  // obtain number of account to delete
  printf("%s", "Enter account number to delete ( 1 - 100 ): ");
  scanf("%d", &accountNum);

  // move file pointer to correct record in file
  fseek(fPtr, (accountNum - 1) * sizeof(struct clientData), SEEK_SET);
  // read record from file
  fread(&client, sizeof(struct clientData), 1, fPtr);
  // display error if record does not exist
  if (client.acctNum == 0) {
    printf("Account %d does not exist.\n", accountNum);
  }      // end if
  else { // delete record
    // move file pointer to correct record in file
    fseek(fPtr, (accountNum - 1) * sizeof(struct clientData), SEEK_SET);
```

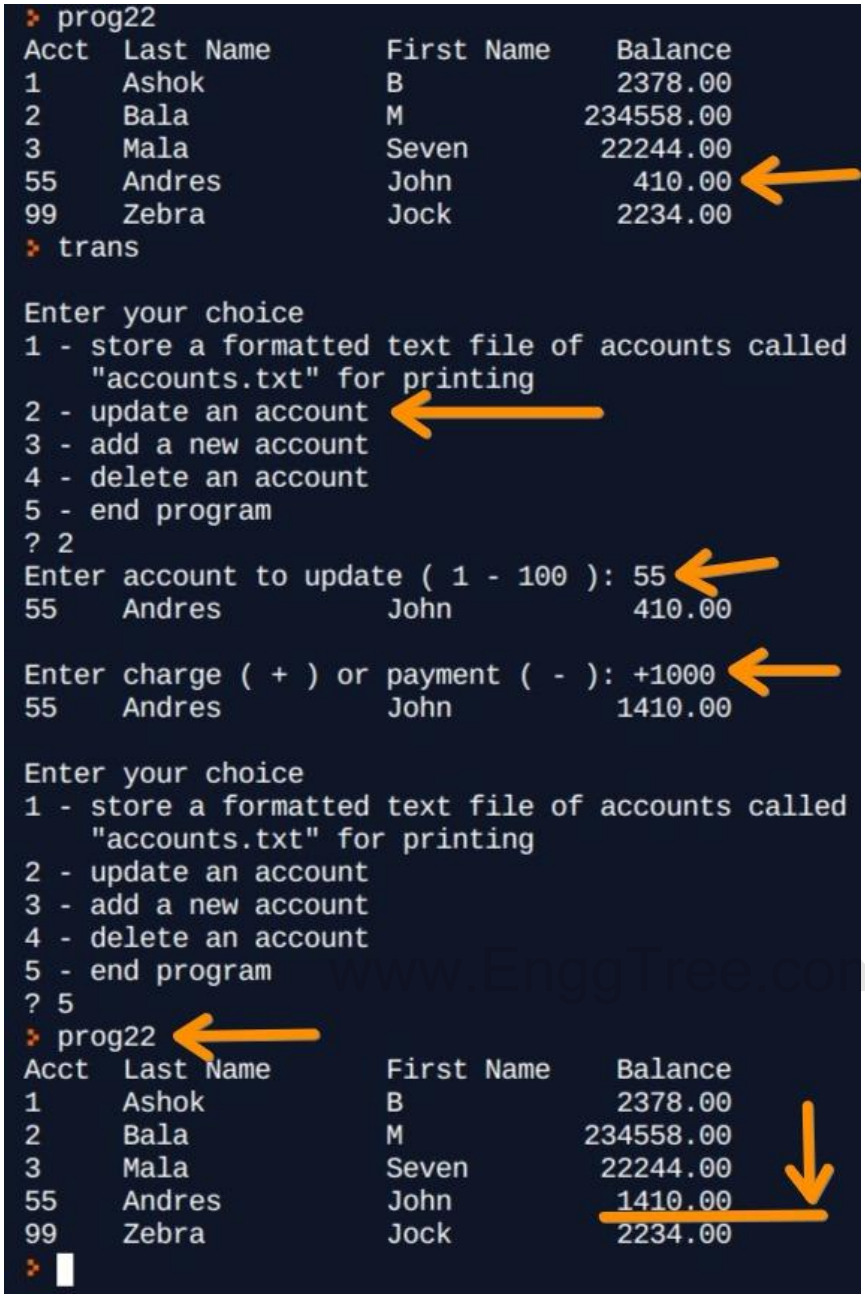
```
    // replace existing record with blank record
    fwrite(&blankClient, sizeof(struct clientData), 1, fPtr);
  } // end else
} // end function deleteRecord
// create and insert record
void newRecord(FILE *fPtr) {
  // create clientData with default information
  struct clientData client = {0, "", "", 0.0};
  unsigned int accountNum; // account number

  // obtain number of account to create
  printf("%s", "Enter new account number ( 1 - 100 ): ");
  scanf("%d", &accountNum);
```



```
  // move file pointer to correct record in file
  fseek(fPtr, (accountNum - 1) * sizeof(struct clientData), SEEK_SET);
  // read record from file
  fread(&client, sizeof(struct clientData), 1, fPtr);
  // display error if account already exists
  if (client.acctNum != 0) {
    printf("Account #%d already contains information.\n", client.acctNum);
  } // end if
  else { // create record user enters last name, first name and balance
    printf("%s", "Enter lastname, firstname, balance\n? ");
    scanf("%14s%9s%lf", client.lastName, client.firstName, &client.balance);
    client.acctNum = accountNum;
```

```
    // move file pointer to correct record in file
    fseek(fPtr, (client.acctNum - 1) * sizeof(struct clientData), SEEK_SET);
    // insert record in file
    fwrite(&client, sizeof(struct clientData), 1, fPtr);
  } // end else
} // end function newRecord

// enable user to input menu choice
unsigned int enterChoice(void) {
  unsigned int menuChoice; // variable to store user's choice
  // display available options
  printf("%s", "\nEnter your choice\n"
          "1 - store a formatted text file of accounts called\n"
          "    \"accounts.txt\" for printing\n"
          "2 - update an account\n"
          "3 - add a new account\n"
          "4 - delete an account\n"
          "5 - end program\n? ");

  scanf("%u", &menuChoice); // receive choice from user
  return menuChoice;
} // end function enterChoice
```

Output



5.7 Command line arguments

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax:

int main(int argc, char *argv[])

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program

Example:

```
#include <stdio.h>

#include <conio.h>

int main(int argc, char *argv[])

{

int i;

if( argc >= 2 )

{

printf("The arguments supplied are:\n");

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

{

printf("%s\t", argv[i]);

}

}

else

{

printf("argument list is empty.\n");

}

return 0;

}
```


Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be 1.

Multiple Choice Questions

1. _____ is a collection of data.

A. Buffer

B. Stream

C. File

Answer: File

2. If the mode includes b after the initial letter, what does it indicates?

a) text file

b) big text file

c) binary file

Answer: binary file

3. What is the function of the mode ‘ w+’?

a) create text file for writing, discard previous contents if any

b) create text file for update, discard previous contents if any

c) create text file for writing, do not discard previous contents if any

d) create text file for update, do not discard previous contents if any

Answer: create text file for update, discard previous contents if any

4. fflush(NULL) flushes all _____

a) input streams

b) output streams

c) previous contents

d) appended text

Answer: output streams

5. **What is the keyword used to declare a C file pointer.?**

A) file

B) FILE

C) FILEFP

D) filefp

Answer: FILE

6. **What is a C FILE data type.?**

A) FILE is like a Structure only

B) FILE is like a Union only

C) FILE is like a user define int data type

D) None of the above

Answer: FILE is like a Structure only

7. **Where is a file temporarily stored before read or write operation in C language.?**

A) Notepad

B) RAM

C) Hard disk

D) Buffer

Answer: Buffer

8. Which function gives the current position of the file.

A. fseek()
B. fsetpos()
C. ftell()
D. Rewind()

Answer: ftell()

9. Which function is used to perform block output in binary files?

A. fwrite()
B. fprintf()

C. fputc()

D. fputs()

Answer: fwrite()

10. Select the standard stream in C

A. stdin

B. stdout

C. stderr

D. all of these

Answer: all of these

11. From which standard stream does a C program read data?

A. Stdin

B. stdout

C. stderr

D. all of these

Answer: stderr

12. Which acts as an interface between stream and hardware?

A. file pointer

B. buffer

C. stdout

D. stdin

Answer: buffer

13. Which function is used to associate a file with a stream?

A. fread()

B. fopen()

C. floes()

D. fflush()

Answer: fopen()

14. Which function returns the next character from stream, EOF if the end of file is reached, or if there is an error?

A. fgetc()

B. fgets()

C. fputs()

D. fwrite()

Answer: fgetc()