

# C Programming & Data Structures

---

## UNIT-1

### Introduction to Computers:

A computer system consists of hardware and software.

**Computer hardware** is the collection of physical elements that comprise a computer system.

**Computer software** is a collection of computer programs and related data that provides the instructions for a computer what to do and how to do it. Software refers to one or more computer programs and data held in the storage of the computer for some purposes.

Basically computer software is of three main types

#### 1. Operating System

The Operating System (OS) is an interface between the compute software and hardware. The most popular and latest operating systems include Windows XP, Mac, UNIX, Linux, Windows Vista, etc.

#### 2. Application Software

The application software is widely used for accomplishment of specific and precise tasks which is a step ahead than the basic operations or running of the computer system. The application software includes printing documents, and permitting access to internet for web and video conferencing activities. The Application software indirectly does the interaction with the machine to perform all these functions.

#### 3. System Software

System software directly interacts with computer hardware. Some of the examples are the device drivers for CPU, Motherboard, Mouse, Printer, Keyboard, etc. The system software takes the responsibility of control, integration and managing individual hardware machine of the computer.

### Computing Environment:

Computing Environment is a collection of computers / machines, software, and networks that support the processing and exchange of electronic information meant to support various types of computing solutions.

#### Types of Computing Environments:

- Personal Computing Environment
- Client Server Environment
- Time sharing Environment
- Distributed Environment

## C Programming & Data Structures

---

### Algorithm:

An algorithm is a description of a procedure which terminates with a result. Algorithm is a step-by-step method of solving a problem.

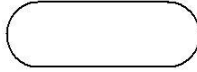

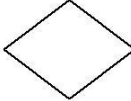
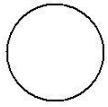

### Properties of an Algorithm:

- 1) Finiteness: - An algorithm terminates after a finite numbers of steps.
- 2) Definiteness: - Each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (explain the meaning of) in multiple ways & can be performed without any confusion.
- 3) Input: - An algorithm accepts zero or more inputs
- 4) Output: - An algorithm should produce at least one output.
- 5) Effectiveness: - It consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

### Flowchart:

A flowchart is a graphical or pictorial representation of an algorithm.

Each step in the process is represented by a different symbol and contains a short description of the process step. The flow chart symbols are linked together with arrows showing the process flow direction. Some common flowcharting symbols are listed below.

	Purpose	Symbol
<b>Terminator</b>	An oval flow chart shape indicating the start or end of the process.	
<b>Process</b>	A rectangular flow chart shape indicating a normal process flow step.	
<b>Decision</b>	A diamond flow chart shape indication a branch in the process flow.	
<b>Connector</b>	A small, labeled, circular flow chart shape used to indicate a jump in the process flow.	
<b>Data</b>	A parallelogram that indicates data input or output (I/O) for a process.	

# C Programming & Data Structures

---

## Program Development Steps:

### 1. Statement of Problem

- a) Working with existing system and using proper questionnaire, the problem should be explained clearly.
- b) What inputs are available, what outputs are required and what is needed for creating workable solution, should be understood clearly.

### 2. Analysis

- a) The method of solutions to solve the problem can be identified.
- b) We also judge that which method gives best results among different methods of solution.

### 3. Design

- a) Algorithms and flow charts will be prepared.
- b) Focus on data, architecture, user interfaces and program components.

### 4. Implementation

The algorithms and flow charts developed in the previous steps are converted into actual programs in the high level languages like C.

#### a. Compilation

The process of translating the program into machine code is called as Compilation. Syntactic errors are found quickly at the time of compiling the program. These errors occur due to the usage of wrong syntaxes for the statements.

Eg:  $x=a*y+b$

There is a syntax error in this statement, since, each and every statement in C language ends with a semicolon (;).

#### b. Execution

The next step is Program execution. In this phase, we may encounter two types of errors. Runtime Errors: these errors occur during the execution of the program and terminate the program abnormally.

Logical Errors: these errors occur due to incorrect usage of the instructions in the program. These errors are neither detected during compilation or execution nor cause any stoppage to the program execution but produces incorrect output.

## C Programming & Data Structures

---

### General Structure of a C program:

```
/* Documentation section
*/ /* Link section */
/* Definition section */
/* Global declaration section
*/ main()
{
    Declaration part
    Executable part (statements)
}
/* Sub-program section */
```

- The documentation section is used for displaying any information about the program like the purpose of the program, name of the author, date and time written etc, and this section should be enclosed within comment lines. The statements in the documentation section are ignored by the compiler.
- The link section consists of the inclusion of header files.
- The definition section consists of macro definitions, defining constants etc.,
- Anything declared in the global declaration section is accessible throughout the program, i.e. accessible to all the functions in the program.
- main() function is mandatory for any program and it includes two parts, the declaration part and the executable part.
- The last section, i.e. sub-program section is optional and used when we require including user defined functions in the program.

# C Programming & Data Structures

## UNIT – II

### C Language Components

The four main components of C language are

- 1) The Character Set.
- 2) Tokens
- 3) Variables
- 4) Data Types

- 1) **The Character Set :** Character set is a set of valid characters that a language can recognize. A character represents any letter, digit or any other sign.

- Letters - A,B,C.....Z or a,b,c .....z.
- Digits - 0,1.....9
- Special Symbols - ~!@#\$%^&.....
- White Spaces - Blank space, horizontal tab, carriage return, new line, form feed.

- 2) **Tokens:** The smallest individual unit in a program is known as a token. C has five tokens

- i. Keywords
- ii. Identifiers
- iii. Constants
- iv. Punctuations
- v. Operators

- i. **Keywords:** Keywords are reserved word in C. They have predefined meaning cannot be changed. All keywords must be written in lowercase. Eg:- auto,long,char,short etc.

- ii. **Identifiers:** - Identifiers refer to the names of variable, functions and arrays. These are user-defined names. An identifier in C can be made up of letters, digits and underscore. Identifiers may start with either alphabets or underscore. The underscore is used to make the identifiers easy to read and mark functions or library members.

- iii. **Constants:** - Constants in C refers to fixed values that do not change during the execution of a program. C support several types of constants.

- a. Numerical Constants
  - i. Integer Constant
    1. Decimal Constant
    2. Octal Constant
    3. Hexadecimal Constant
  - ii. Float Constant

## C Programming & Data Structures

---

- b. Character Constants
  - i. Single Character Constant
  - ii. String Constant

**Integer Constant:** - An integer constant is a whole number without any fractional part. C has three types of integer constants.

**Decimal Constant:** - Decimal integers consists of digits from 0 through 9  
Eg.: 34,900,3457,-978

**Octal Constant:** - An Octal integer constant can be any combination of digits from 0 through 7. In C the first digit of an octal number must be a zero(0) so as to differentiate it from a decimal number. Eg.: 06,034,-07564

**Hexadecimal Constant:** Hexadecimal integer constants can be any combination of digits 0 through 9 and alphabets from „a“ through „f“ or „A“ through „F“. In C, a hexadecimal constant must begin with 0x or 0X (zero x) so as to differentiate it from a decimal number. Eg:- 0x50,0XAC2 etc

**Floating Constants (Real):** Real or floating point numbers can contain both an integer part and a fractional part in the number. Floating point numbers may be represented in two forms, either in the fractional form or in the exponent form.

A float point number in fractional form consists of signed or unsigned digits including decimal point between digits. E.g:- 18.5, .18 etc.

Very large and very small numbers are represented using the exponent form. The exponent notation use the „E“ or „e“ symbol in the representation of the number. The number before the „E“ is called as mantissa and the number after forms the exponent.

Eg.: -5.3E<sup>-5</sup>, -6.79E<sup>3</sup>, 78e<sup>05</sup>

**Single Character Constant:** - A character constant is usually a single character or any symbol enclosed by apostrophes or single quotes. Eg.: ch=„a“

**String Constant:** - A sequence of character enclosed between double quotes is called string constant. Eg.: „Hello Good Morning“

**iv) Punctuations:** - 23 characters are used as punctuations in C. eg: + \_ / ; : > ! etc

**v) Operators:** - An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation on data stored in variables. The variables that are operated as operands. C operator can be classified into 8 types.

- i. Arithmetic Operators : + - \* / %
- ii. Assignment Operators : =
- iii. Relational Operators: < > <= >= == !=
- iv. Logical Operators: ! && ||
- v. Conditional Operators: ! :
- vi. Increment & Decrement Operator : ++ --
- vii. Bitwise Operator: ! & | ~ ^ << >>
- viii. Special Operator : sizeof ,(comma)

## C Programming & Data Structures

---

- 3) **Variables:** A variable is an object or element that may take on any value or a specified type. Variable are nothing but identifiers, which are used to identify variables programming elements. Eg: name, sum, stu\_name, acc\_no etc.
- 4) **Data types:** Data types indicate the types of data a variable can have. A data types usually define a set of values, which can be stored in the variable along with the operations that may be performed on those values. C includes two types of data.
- **Simple or Fundamental data type**
  - **Derived data type**

**Simple Data Types:** There are four simple data types in C.

- **int**
- **char**
- **float**
- **double**

**int:-** This means that the variable is an integer are stored in 2 bytes, may range from -32768 to 32767.

**char:-** This means that the variable is a character type, char objects are stored in one byte. If unsigned, the values may be in the range 0 to 255.

**Float:-** This means that the variable is a real number stored in 4 bytes or 32 bits. The range of floating point values are between  $3.4E^{-38}$  to  $3.4E^{38}$  or 6 significant digits after the decimal point.

**Double:** This means that the variable is a double precision float type. In most cases the systems allocates 8 bytes or 64 bits space, between  $1.7E^{-308}$  to  $1.7E^{308}$ .

**Derived Data Types:** Derived data types are constructed from the simple data types and or other derived data types. Derived data include arrays, functions, pointers, references, constants, structures, unions and enumerations.

Variable Type	Keyword	Bytes Required	Range
Character	Char	1	-128 to 127
Unsigned character	unsigned char	1	0 to 255
Integer	Int	2	-32768 to +32767
Short integer	short int	2	-32768 to 32767
Long integer	long int	4	-2147483648 to 2147483647
Unsigned integer	unsigned int	2	0 to 65535
Unsigned short integer	unsigned short int	2	0 to 65535

## C Programming & Data Structures

Unsigned long integer	unsigned long int	4	0 to 4294967295
Float	Float	4	3.4E +/- 38 (7 digits)
Double	Double	8	1.7E +/- 308 (15 digits)
Long Double	long double	10	3.4E-4932 to 1.1E+4932

### Precedence of Operators:

Precedence is defined as the order in which the operators in a complex evaluation are evaluated.

### Associativity:

When two or more operators have the same precedence then the concept of associativity comes into discussion.

Operator	Description	Level	Associativity
[ ] . ( ) ++ --	access array element access object member invoke a method post-increment post-decrement	1	left to right
++ -- + - ! ~	pre-increment pre-decrement unary plus unary minus logical NOT bitwise NOT	2	right to left
() new	Cast object creation	3	right to left
* / %	multiplicative	4	left to right
+ - +	Additive string concatenation	5	left to right
<<>> >>>	Shift	6	left to right
< <= > >= instanceof	Relational type comparison	7	left to right
== !=	Equality	8	left to right
&	bitwise AND	9	left to right
^	bitwise XOR	10	left to right
	bitwise OR	11	left to right



## C Programming & Data Structures

&&	conditional AND	12	left to right
	conditional OR	13	left to right
?:	conditional	14	right to left
= += -= *= /= %= &= ^=  = <<= >>= >>>=	assignment	15	right to left

Precedence and Associativity of various operators

### Type Conversion

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a "narrower" operand into a "wider" one without losing information, such as converting an integer into floating point in an expression like `f + i`. Expressions that don't make sense, like using a float as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

A char is just a small integer, so chars may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function `atoi`, which converts a string of digits into its numeric equivalent.

Implicit arithmetic conversions work much as expected. In general, if an operator like `+` or `*` that takes two operands (a binary operator) has operands of different types, the "lower" type is *promoted* to the "higher" type before the operation proceeds. The result is of the integer type. If there are no unsigned operands, however, the following informal set of rules will suffice:

- ⤴ If either operand is long double, convert the other to long double.
- ⤴ Otherwise, if either operand is double, convert the other to double.
- ⤴ Otherwise, if either operand is float, convert the other to float.
- ⤴ Otherwise, convert char and short to int.
- ⤴ Then, if either operand is long, convert the other to long.

### Input-output in C:

**Input:** In any programming language input means to feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

**Output:** In any programming language output means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output required data.

There are two types of I/O statements in C. They are *unformatted I/O functions* and *formatted I/O functions*.

## C Programming & Data Structures

---

### Unformatted I/O functions:

- 1.getchar():Used to read a character
- 2.putchar():Used to display a character
- 3.gets():Used to read a string
- 4.puts():Used to display a string which is passed as argument to the function

### Formatted I/O functions:

printf() and scanf() are examples of formatted I/O functions.

printf() is an example of formatted output function and scanf() is an example of formatted input function.

### Conditional Statements:

Conditional statements control the sequence of statement execution, depending on the value of a integer expression.

#### 1) If Statement

- a)Simple If statement
- b) If-else statement
- c) if-else if statement
- d) Nested if statement

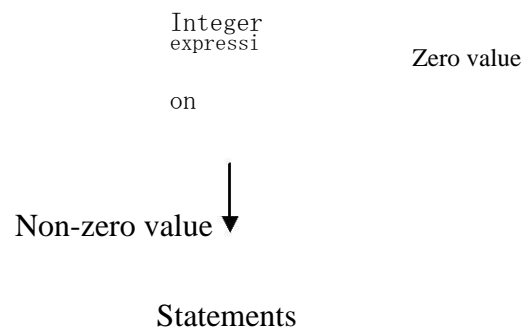
#### 2) Switch Statement

##### 1)If Statement:

- i) Simple if

Syntax:

```
if ( <integer expression> )  
    <statement list>
```



## C Programming & Data Structures

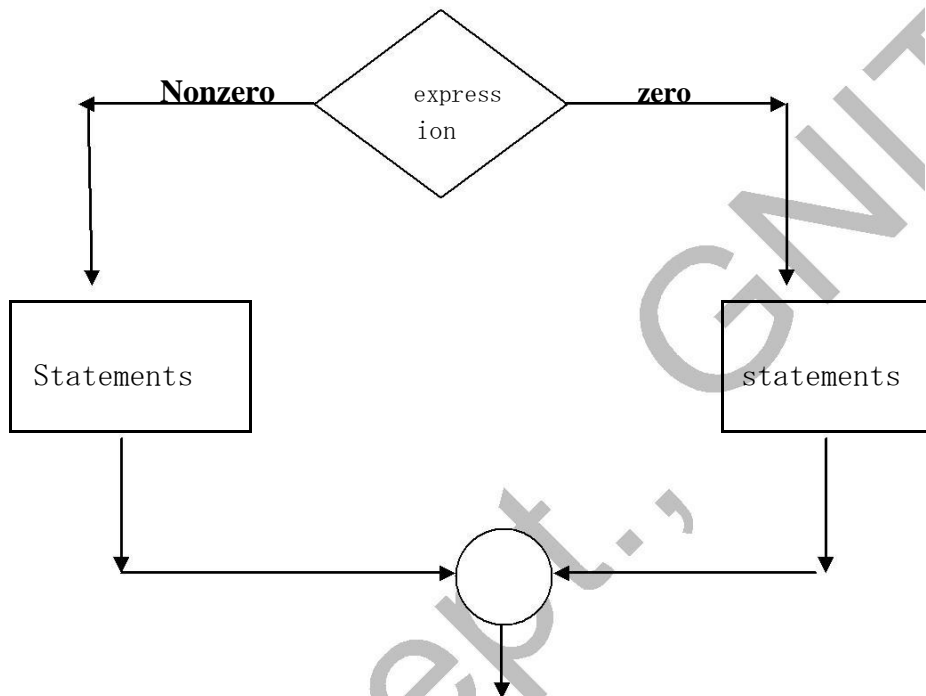
### ii) If-else statement

Syntax:

```

if ( <integer expression> )
{ <statement list>
} else {
<statement list>
}

```



### iii) if-else if statement

Syntax:

```

if ( <integer expression> )
{ <statement list>
} else if ( <integer expression> ) {
<statement list>
...
} else if ( <integer expression> ) {
<statement list>
} else {
<statement list>
}

```

## C Programming & Data Structures

### iv) Nested if statement

```

if ( <integer expression> ) { if(<integer expression>{
<statement list> }else{<statement list>}
} else { if(<integer expression>{
<statement list> }else{<statement list>}
}

```

### Switch statement:

The switch statement provides a multi-way selection control structure to execute a selected section of code determined by the run-time value of an expression. The condition is an expression which evaluates to an integer value, which include signed, unsigned, char, enum, and even boolean values.

```

Syntax:      switch ( <expression> ) {
                case this-value:
                    Code to execute if <expression> == this-value
                break;
                case that-value:
                    Code to execute if <expression> == that-value
                break;
                ...
                default:
                    Code to execute if <expression> does not equal the value following
                    any of the cases
                break;
            }

```

### Sample program:

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

```

main()
{
    int menu, numb1, numb2, total;

    printf("enter in two numbers -->");
    scanf("%d %d", &numb1, &numb2 );
    printf("enter in choice\n");
    printf("1=addition\n");
    printf("2=subtraction\n");
    scanf("%d", &menu );
    switch( menu ) {
        case 1: total = numb1 + numb2; break; case
        2: total = numb1 - numb2; break; default:
        printf("Invalid option selected\n");
    }
}

```

## C Programming & Data Structures

---

```
        if( menu == 1 )
            printf("%d plus %d is %d\n", numb1, numb2, total );
        else if( menu == 2 )
            printf("%d minus %d is %d\n", numb1, numb2, total );
    }
```

### Sample Program Output

```
enter in two numbers --> 37
23 enter in choice
1=addition
2=subtraction
2
37 minus 23 is 14
```

### Looping Statements:

Loops provide a way to repeat a set of statements and control how many times they are repeated.

C supports three looping statements. They are **while**, **do-while** and **for**.

Both while and for statements are called as **entry-controlled** loops because they evaluate the expression and based on the value of the expression they transfer the control of the program to a particular set of statements.

Do-while is an example of **exit-controlled** loop as the body of the loop will be executed once and then the expression is evaluated. If it is non-zero value then the body of the loop will be executed once again until the expression's value becomes zero.

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming -- many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

### While loop:

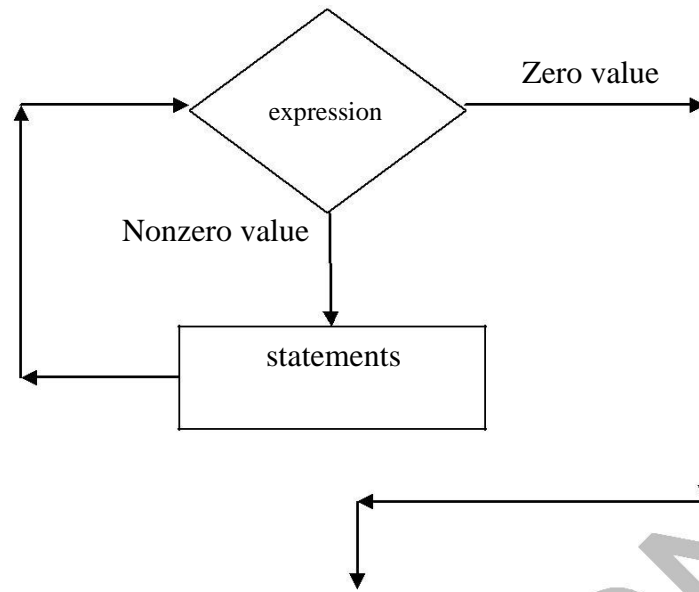
This is an entry controlled looping statement. It is used to repeat a block of statements until the expression evaluates to nonzero value

#### Syntax:

```
while(expression) {
    statements;
    increment/decrement;
}
```

#### Flowchart for while:

## C Programming & Data Structures



Example program for while:

```

#include<stdio.h> int
main()
{
    int i,times;
    scanf("%d",&times);
    i = 0;
    while (i <= times)
    {
        i++;
        printf("%d\n",i);
    }
    return 0;
}
  
```

OUTPUT:

3  
1  
2  
3

**For loop:**

This is an entry controlled looping statement.

In this loop structure, more than one variable can be initialized. One of the most important features of this loop is that the three actions can be taken at a time like variable initialization, condition checking and increment/decrement. The for loop can be more concise and flexible than that of while and do-while loops.

## C Programming & Data Structures

---

### Syntax:

```
for(initialization; test-condition; increment/decrement)
{
    statements;
}
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the loop and executes the block of statements associated with it. At the end of loop increment or decrement is done to change in variable value. This process continues until test condition satisfies.

### **Do-While loop:**

This is an exit controlled looping statement.

Sometimes, there is need to execute a block of statements first then to check condition. At that time such type of a loop is used. In this, block of statements are executed first and then condition is checked.

### Syntax:

```
do
{
    statements;
    (increment/decrement);
} while (expression);
```

In above syntax, the first the block of statements are executed. At the end of loop, while statement is executed. If the expression is evaluated to nonzero value then program control goes to evaluate the body of a loop once again. This process continues till expression evaluates to a zero. When it evaluates to zero, then the loop terminates.

### **Break and continue statements:**

C break statement is used to terminate any type of loop such as while loop, do while loop and for loop. C break statement terminates the loop body immediately and passes control to the next statement after the loop.

C continue statement is used to skip over the rest of the current iteration. After continue statement, the control returns to the top of the loop.

### Example program for continue:

```
main()
{
    int i;
    int j = 10;
    for( i = 0; i <= j; i ++ )
    {
        if( i == 5 )
        {
            continue;
        }
    }
}
```

## C Programming & Data Structures

---

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

OUTPUT:

```
Hello 0  
Hello 1  
Hello 2  
Hello 3  
Hello 4  
Hello 6  
Hello 7  
Hello 8  
Hello 9  
Hello 10
```

### **Goto statement:**

A goto statement is used to branch (transfer control) to another location in a program. Syntax:

```
goto label;
```

The label can be any valid identifier name and it should be included in the program followed by a colon.



# C Programming & Data Structures

---

## UNIT - III

### FUNCTIONS

**Definition:** Modules in C are called functions. A function in C is defined to be the program segment that carries out some specific, well defined task. A function is a block of code that has a name and it has a property that it is reusable i.e. it can be executed from as many different points in a C Program as required.

There are two types of functions:

**Library functions:** C standard library provides a rich collection of functions for performing I/O operations, mathematical calculations, string manipulation operations etc. For example, *sqrt(x)* is a function to calculate the square root of a number provided by the C standard library and included in the <math.h> header file.

Note that each program in C has a function called *main* which is used as the root function of calling other library functions.

**Programmer Defined functions:** In C, the programmers can write their own functions and use them in their programs.

The user defined function can be explained with three elements.

1. Function definition
2. Function call
3. Function declaration

The function definition is an independent program module that is specially written to implement the requirements of the function. In order to use this function we need to invoke it at a required place in the program. This is known as function call. The program that calls the function is known as calling program or calling function. The calling program should declare any function that is to be used later in the program. This is known as the function declaration or the function prototype.

#### Structure of a function:

There are two main parts of the function. The function header and the function body.

Consider the following

```
example: int sum(int x, int y)
{
    int ans = 0;
    ans = x + y;
    return ans
}
```

#### Function Header

In the first line of the above code int sum(int x, int y)

It has three main parts

## C Programming & Data Structures

---

1. The name of the function i.e. *sum*
2. The parameters of the function enclosed in paranthesis
3. Return value type i.e. *int*

### Function Body

Whatever is written with in { } in the above example is the body of the function.

**Function prototype:** Function prototypes are always declared at the beginning of the program indicating the name of the function, the data type of its arguments which is passed to the function and the data type of the returned value from the function.

**Ex: int square( int);**

The prototype of a function provides the basic information about a function which tells the compiler that the function is used correctly or not. It contains the same information as the function header contains. The prototype of the function in the above example would be like

`int sum (int x, int y);`

The only difference between the header and the prototype is the semicolon ; there must the a semicolon at the end of the prototype.

### Categories of functions:

A function depending on whether arguments are present or not and whether value is returned or not, may belong to one of the following categories:

1. Functions with no arguments and no return values
2. Functions with arguments and no return values
3. Functions with arguments and a return value
4. Functions with no arguments but a return value

### Example programs:

*/\* The program illustrates the **functions with no arguments** and no return value\*/*

```
#include<stdio.h>
void add(void);
void main()
{
    add();
}
void add(void)
{
    int x,y,sum;
    printf("Enter any two integers:");
    scanf("%d%d",&x,&y);
    sum=x+y;
    printf("The sum id %d",sum);
}
```

Output:

```
Enter any two integers: 2
4 The sum is 6
```

## C Programming & Data Structures

---

/\* The program illustrates the **functions with arguments** and no return value\*/

```
#include<stdio.h>
void add(int a,int
b); void main()
{
int x,y;;
printf("Enter any two integers:");
scanf("%d%d",&x,&y); add(x,y);
```

```
}
void add(int a,int b)
{
int sum;
sum=a+b;
printf("The sum id %d",sum);
}
```

Output:

Enter any two integers: 2  
4 The sum is 6

/\* The program illustrates the **functions with arguments and a return value**\*/

```
#include<stdio.h>
int add(int a,int b);
void main()
{
int x,y,sum;
printf("Enter any two integers:");
scanf("%d%d",&x,&y);
sum=add(x,y);
printf("The sum id %d",sum);
}
```

```
int add(int a,int b)
{
int c;
c=a+b;
return c;
}
```

Output:

Enter any two integers: 2  
4 The sum is 6

## C Programming & Data Structures

---

/\* The program illustrates the **functions with no arguments and but a return value\***\*/

```
#include<stdio.h>
int add(void); void
main()
{
int sum;
sum=add();
printf("The sum id %d",sum);
}
void add(void)
{
int x,y,c;
printf("Enter any two integers:");
scanf("%d%d",&x,&y);
c=x+y;
return c;
}
```

**Output:**

Enter any two integers: 2

4 The sum is 6

### **Inter function Communication:**

Although the calling and called functions are two separate entities, they need to communicate to exchange data. The data flow between the calling and called functions can be divided into three strategies:

1. A downward flow from the calling to the called function,
2. An upward flow from the called to the calling function
3. A bidirectional flow from both the directions.

### **Downward flow:**

Here the calling function sends data to the called function. No data flows in the opposite direction.

### **Example program:**

```
void downfun(int x,int
y); int main()
{
int a=5;
downfun(a,15);
printf("%d",a);
return 0;
}
void downflun(int x, int y)
{
x=x+y;
return;
}
```

## C Programming & Data Structures

---

### Upward flow:

This occurs when the called function sends data back to the called function without receiving any data from it.

### Example program:

```
void upfun(int *ax,int
*ay); int main(void)
{
int a;
int b;
upfun(&a,&b);
printf("%d
%d\n",a,b); return 0;
}
void upfun(int *ax,int *ay)
{
*ax=23;
*ay=8;
return;
}
```

### Bidirectional flow:

This occurs when the calling function sends data down to the called function. During or at the end of its processing, the called function then sends data up to the calling function.

### Types of function calls

**Call by Value:** When a function is called by an argument/parameter the copy of *the argument* is passed to the function. If the argument is a normal (non-pointer) value a possible change on the copy of the argument in the function does not change the original value of the argument. However, given a pointer/address value any change to the value pointed by the pointer/address changes the original argument.

**Ex:** The following program is to calculate and print the area and the perimeter of a circle. by using *Call by value* approach

```
#include<stdio.h>/
#define pi 3.14
float area(float);
float perimeter(float);
int main( )
{
float r, a, p;
printf("Enter the radius\n");
scanf("%f",&r);
a = area(r);
p = perimeter(r);
printf("The area = %.2f, \n The Perimeter = %.2f", a,
p); return 0;
}
```

## C Programming & Data Structures

---

```
float area(float x)
{
    return pi*r*r;
}
float perimeter(float y)
{
    return 2.0*pi*r;
}
```

**Call by Reference:** When a function is called by an argument/parameter which is a pointer(address of the argument) the copy of *the address of the argument* is passed to the function. Therefore a possible change on the data at the referenced address change the original value of the argument.

**Ex:** The following program is to calculate and print the area and the perimeter of a circle. by using

*Call by reference* approach

```
#include<stdio.h>
#define pi 3.14
void area_perimeter(float, float *, float
*); int main( )
{
    float r, a, p;
    printf("Enter the
radius\n"); scanf("%f",&r);
    area_perimeter(r,&a,&p);
    printf("The area = %.2f, \n The Perimeter = %.2f", a,
p); return 0;
}
void area_perimeter(float x, float *aptr, float *pptr);
{
    *aptr = pi*x*x;
    *pptr = 2.0*pi*x;
}
```

### Recursion:

Recursion is a process in which a function calls itself.

**Example:**

```
Recursion()
{
    printf("Recursion
!"); Recursion();
}
```

## C Programming & Data Structures

---

### **Program :**

*/\* Program to demonstrate recursion \*/.*

```
#include <stdio.h>
int fact(int x); void
main()
{
    int a,f;
    clrscr();
    printf("Enter any
integer:"); scanf("%d",&a);
    f=fact(a);
    printf("The factorial of %d is %d",a,f);
}
int fact(int x)
{
    if(x==0)
        return 1;
    else
        return(x*fact(x-1));
}
```

Output:

Enter any integer:5

The factorial of 5 is 120

### **Features :**

- There should be at least one if statement used to terminate recursion.
- It does not contain any looping statements.

### **Advantages :**

- It is easy to use.
- It represents compact programming structures.

### **Disadvantages :**

- It is slower than that of looping statements because each time function is called.

### **Standard Functions:**

C provides a rich collection of standard functions whose definitions have been written and are ready to be used in our programs.

- Absolute value function:
- An absolute value is the positive rendering of the value regardless of its sign.
- abs(3) returns 3
- Ceiling function:

## C Programming & Data Structures

---

- A ceiling is the smallest integral value greater than or equal to a number.
- `ceil(3.001)=4`
- Floor function:
- `floor(1.2)` returns 1.0
- Truncate function:
- The truncate functions return the integral in the direction of 0. They are the same as the floor function for positive numbers and the same as ceiling function for negative numbers.
- `trunc(-1.1)` returns -1.0
- `trunc(1.9)` returns 1.0
- Round function:
- The round functions return the nearest integral value.
- `round(1.9)=2.0`
- Power function:
- The power (`pow`) function returns the value of the raised to the power `y`.
- `pow(3.0,4.0)` return 81.0
- Square root function:
- The square root functions return the non negative square root of a number. An error occurs if the number is negative.
- `sqrt(25)` returns 5.0

### The Preprocessor Directives

The C preprocessor provides several tools that are not available in other high-level languages. The programmer can use these tools to make his program more efficient in all aspects.

#### Working of preprocessor

When a command is given to compile a C program, the programs run automatically through the preprocessor. The preprocessor is a program that modifies the C source program according to directives supplied in the program. An original source program usually is stored in a file. The preprocessor does not modify this program file, but creates a new file that contains the processed version of the program. This new file is then submitted to the compiler. Some compilers enable the programmer to run only the preprocessor on the source program and to view the results of the preprocessor stage.

All preprocessor directives begin with the number or sharp sign (`#`). They must start in the first column, and no space is required between the number sign and the directive. The directive is terminated not by a semicolon, but by the end of the line on which it appears.

The C preprocessor is a collection of special statements called directives that are executed at the beginning of the compilation process. The `#include` and `#define` statements are preprocessor directives.

#### # define DIRECTIVE

The `#define` directive is used to define a symbol to the preprocessor and assign it a value. The symbol is meaningful to the preprocessor only in the lines of code following the definition. For example, if the directive `#define NULL 0` is included in the program, then in all lines following the definition, the symbol `NULL` is replaced by the symbol. If the symbol `NULL` is written in the program before the definition is encountered, however, it is not replaced. The `#define` directive is



## C Programming & Data Structures

---

followed by one or more spaces or tabs and the symbol to be defined. The syntax of a preprocessor symbol is the same as that for a C variable or function name. It cannot be a keyword or a variable name used in the program; if it is so a syntax error is detected by the compiler.

### Constants

A common use for defined symbols is the implementation of named constants. The following are examples of constants in C:

23, „a“, “hello”

Any of these can be assigned to a defined preprocessor

symbol: `#define INTEGER 23`

`#define CHARACTER „a“`

A defined symbol can specify only a complete constant.

### MACROS

When a constant is associated with a defined symbol, the characters making up the constants are assigned, not the numeric value of the constant. The definition

`#define EOF -1` really assigns the string “-1” to EOF. The preprocessor replaces the

symbol EOF by the characters “-1” without any consideration of the meaning of the two characters.

When a preprocessor symbol is defined as a segment of text, it is

more generally called a macro. Macros are very useful in making C code more readable and compact. For example, some programmers include the following definitions in all their programs:

#### The # undef Directive:-

If a preprocessor symbol has already been defined, it must be undefined before being redefined. This is accomplished by the `#undef` directive, which specifies the name of the symbol to be undefined. It is not necessary to perform the redefinition at the beginning of a program. A symbol can be redefined in the middle of a program so that it has one value in the first part and another value at the end. A symbol need not be redefined after it is undefined.

#### The #include Directive:-

Often a programmer accumulates a collection of useful constants and macro definitions that are used in almost every program. It is desirable to be able to store these definitions in a file that can be inserted automatically into every program. This task is performed by the `#include` directive.

## Storage Classes

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

**Scope:** The scope of variable determines over what region of the program a variable is actually available for use.

**Visibility:** The program’s ability to access a variable from the memory.

**Lifetime:** Life time refers to the period during which a variable retains a given value during the execution of a program.

## C Programming & Data Structures

---

### Scope rules:

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.\
3. The scope of a formal function argument is its own function.
4. The life time of an auto variable declared in main is the entire program execution time, although its scope is only the main function.
5. The life of an auto variable declared in a function ends when the function is exited.
6. All variables have visibility in their scope , provided they are not declared again.
7. A variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

These are following storage classes which can be used in a C Program:

- auto
- register
- static
- extern

### auto - Storage Class

**auto** is the default storage class for all local variables and the local variable is known only to the function in which it is declared. If the value is not assigned to the variable of type auto the default value is garbage value.

### Syntax :

```
auto [data_type] [variable_name];
```

### Example :

```
auto int a;
```

### Program :

```
/* Program to demonstrate automatic storage class.*/
```

```
#include <stdio.h>
#include <conio.h>
void main()
{
    auto int i=10;
    clrscr();
    {
        auto int i=20;
        printf("\n\t %d",i);
    }
    printf("\n\n\t %d",i);
}
```

## C Programming & Data Structures

---

```
    getch();
}
```

### Output :

```
20
10
```

### register - Storage Class

**Register** is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location). If the value is not assigned to the variable of type register the default value is garbage value.

### Syntax :

```
register [data_type] [variable_name];
```

### Example :

```
register int a;
```

When the calculations are done in CPU, the value of variables are transferred from main memory to CPU. Calculations are done and the final result is sent back to main memory. This leads to slowing down of processes. Register variables occur in CPU and value of that register variable is stored in a register within that CPU. Thus, it increases the resultant speed of operations. There is no waste of time, getting variables from memory and sending it to back again. It is not applicable for arrays, structures or pointers. It cannot be used with static or external storage class.

### Program :

```
/* Program to demonstrate register storage class.*/
```

```
#include <stdio.h>
#include <conio.h>
void main()
{
    register int
    i=10; clrscr();
    {
        register int i=20;
        printf("\n\t %d",i);
    }
    printf("\n\n\t
    %d",i); getch();
}
```

## C Programming & Data Structures

---

**Output :**

20  
10

**static - Storage Class**

**static** is the default storage class for global variables. As the name suggests the value of static variables persists until the end of the program. Static can also be defined within a function. If this is done the variable is initialised at run time but is not reinitialized when the function is called. This inside a function static variable retains its value during various calls. If the value is not assigned to the variable of type static the default value is zero.

**Syntax :**

```
static [data_type] [variable_name];
```

**Example :**

```
static int a;
```

There are two types of static variables:

- a) Local Static Variable
- b) Global Static Variable

Static storage class can be used only if we want the value of a variable to persist between different function calls.

**Program :**

```
/* Program to demonstrate static storage class. */  
#include <stdio.h>  
#include <conio.h>  
void main()  
{  
    int i;  
    void incre(void);  
    clrscr();  
    for (i=0; i<3;  
        i++) incre();  
    getch();  
}  
void incre(void)  
{  
    int avar=1;  
    static int  
    svar=1; avar++;  
    svar++;
```

## C Programming & Data Structures

---

```
    printf("\n\n Automatic variable value : %d",avar);
    printf("\t Static variable value : %d",svar);
}
```

**Output :**

Automatic variable value : 2 Static variable value : 2  
Automatic variable value : 2 Static variable value : 3  
Automatic variable value : 2 Static variable value : 4

**extern - Storage Class**

Variables that are both alive and active throughout the entire program are known as external variables. **extern** is used to give a reference of a global variable that is visible to all the program files. If the value is not assigned to the variable of type static the default value is zero.

**Syntax :**

```
extern [data_type] [variable_name];
```

**Example :**

```
extern int a;
```

The variables of this class can be referred to as 'global or external variables.' They are declared outside the functions and can be invoked at anywhere in a program.

**Program :**

```
/* Program to demonstrate external storage class.*/
#include <stdio.h>
#include <conio.h>
extern int i=10;
void main()
{
    int i=20;
    void show(void);
    clrscr();
    printf("\n\t
    %d",i); show();
    getch();
}
void show(void)
{
    printf("\n\n\t %d",i);
}
```

**Output :**

20  
10

## C Programming & Data Structures

---

### Type qualifiers:

The type qualifier adds three special attributes to types: **const**, **volatile** and **restrict**.

### Constants:

The keyword for the constant type qualifier is **const**. A constant object must be initialized when it is declared because it cannot be changed later. A simple constant is shown below:

```
const double PI=3.1415926
```

### Volatile:

The volatile qualifier tells the computer that an object value may be changed by entities other than this program.

```
volatile int x;
```

### Restricted:

The restrict qualifier which is used only with pointers, indicates that the pointer is only the initial way to access the dereferenced data.

```
restrict int *ptr;
```

## C Programming & Data Structures

---

### UNIT IV

#### ARRAYS

**Definition:**

An array is a collective name given to a group of similar elements. These similar elements could be all integers or all floats or all characters etc.

Usually, the array of characters is called a “string”, where as an array of integers or floats is called simply an array. All elements of any given array must be of the same type i.e we can’t have an array of 10 numbers, of which 5 are ints and 5 are floats.

**Declaration of an Array**

Arrays must be declared before they can be used in the program. Standard array declaration is

as: `type variable_name[lengthofarray];`

Here type specifies the variable type of the element which is going to be stored in the

array. Ex: `double height[10];`

`float width[20];`

In C Language, array starts at position 0. The elements of the array occupy adjacent locations in memory. C Language treats the name of the array as if it was a pointer to the first element This is important in understanding how to do arithmetic with arrays. Any item in the array can be accessed through its index, and it can be accessed any where from within the program. So

`m=height[0];`

variable m will have the value of first item of array height.

**Initializing Arrays**

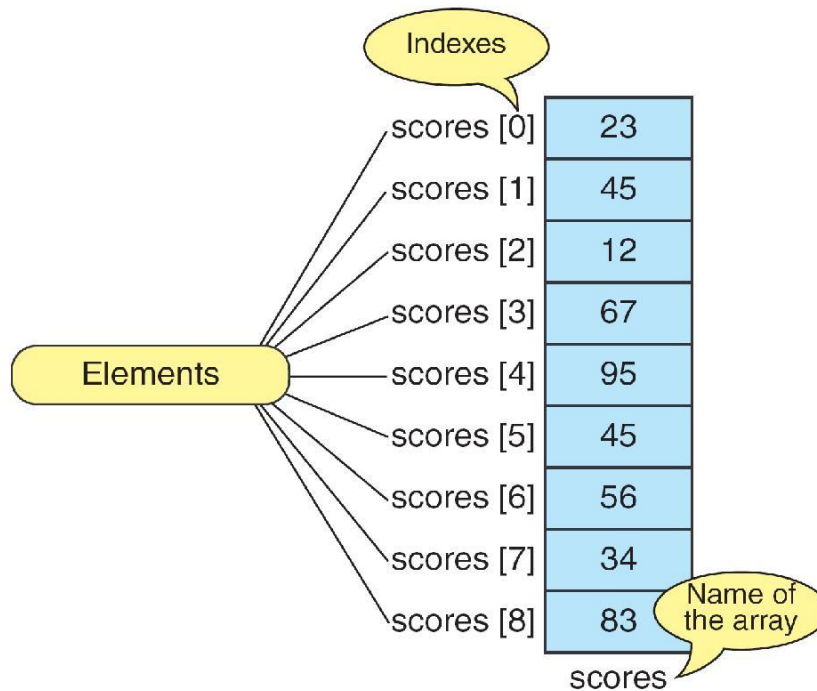
The following is an example which declares and initializes an array of nine elements of type int. Array can also be initialized after declaration.

`int scores[9]={23,45,12,67,95,45,56,34,83};`

---

## C Programming & Data Structures

---



Copy one array into another:

There is no such statement in C language which can directly copy an array into another array. So we have to copy each item separately into another array. The following program illustrates the copying of elements of one array to another array:

```
#include <stdio.h>
int main()
{
    int iMarks[4];
    short newMarks[4];
    iMarks[0]=78;
    iMarks[1]=64;
    iMarks[2]=66;
    iMarks[3]=74;
    for(i=0; i<4; i++)
        newMarks[i]=iMarks[i];
    for(j=0; j<4; j++)
        printf("%d\n", newMarks[j]);
    return 0;
}
```

To summarize, arrays provide a simple mechanism where more than one elements of same type are to be used. We can maintain, manipulate and store multiple elements of same type in one array variable and access them through index.



## C Programming & Data Structures

### Two dimensional arrays

C allows us to define the table of items by using two dimensional arrays.

#### Declaration:

Two dimensional arrays are declared as follows:

```
.           type array_name[row_size][column_size];
```

#### Initialization

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

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

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

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

The following is a sample program that stores roll numbers and marks obtained by a student side by side in matrix

```
main ( )
{
int stud [4] [2];
int i, j;
for (i =0; i < =3; i ++ )
{
printf ("\n Enter roll no. and marks");
scanf ("%d%d", &stud [i] [0], &stud [i] [1] );
}
for (i = 0; i < = 3; i ++ ) printf ("\n %d %d",
stud [i] [0], stud [i] [1]);
}
```

### Multidimensional Arrays

C allows arrays of three or more dimensions. The general form of multidimensional array is `type array_name[s1][s2].....[s3];`

where  $s_i$  is the size of  $i$ th

dimension.. Ex: `int survey[3][5][12];`

survey is a 3 dimensional array.

### Inter function communication:

#### Passing arrays through functions:

Like the values of simple variables, it is also possible to pass the values of an array to a function. To pass an array to a called function, it is sufficient to list the name of the array, without any subscripts and the size of the array as arguments.

Ex: `largest(a,n);`

will pass all the elements contained in the array `a` of size `n`. The called function expecting this call must be appropriately defined.

Example

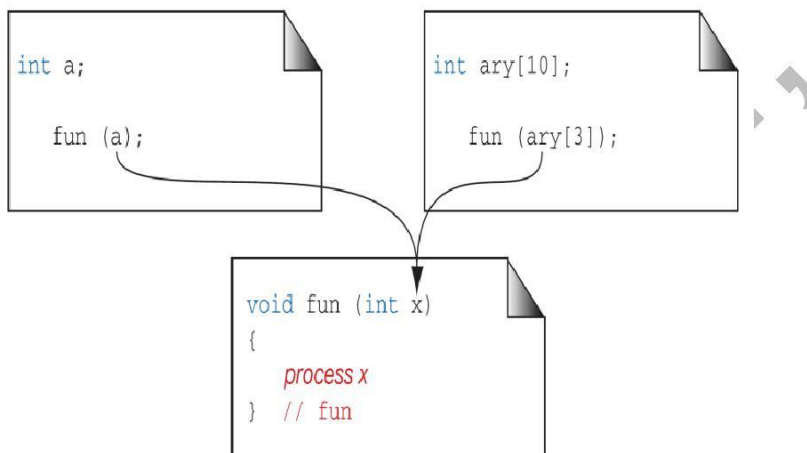
```
program: main()
{
float largest();
```

## C Programming & Data Structures

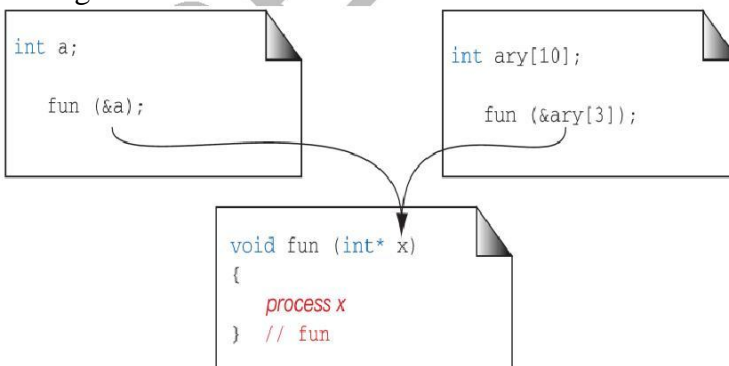
```
static float value[4]={ 10,-4,2,-3};
printf("%f",largest(value,4));
}
float largest(a,n)
float a[];
int n;
{
    int i; float
    max;
    max=a[0];
    for(i=1;i<n;i++)
        if(max<a[i])
            max=a[i];
    return(max);
}
```

To process arrays in a large program, we have to be able to pass them to functions. We can pass arrays in two ways: pass individual elements or pass the whole array.

### Passing array elements

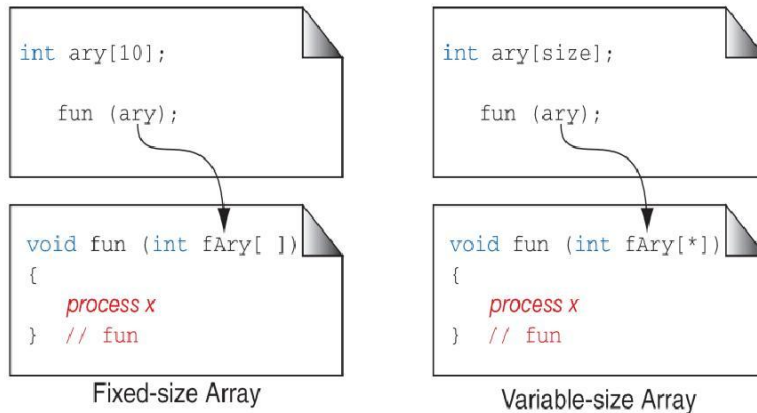


### Passing address of elements



### Passing the whole array

## C Programming & Data Structures



### Array of Strings

Consider, for example, the need to store the days of the week in their textual format. We could create a two-dimensional array of seven days by ten characters, but this wastes space

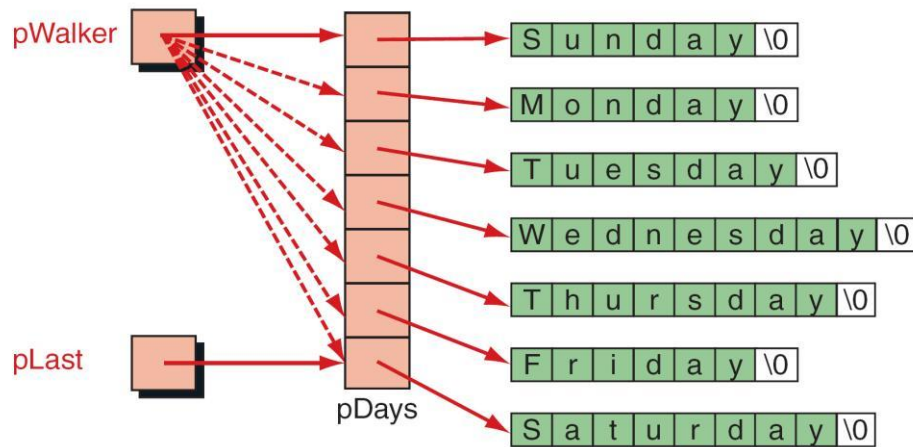
Program to print Days of the Week

```

1  /* Demonstrates an array of pointers to strings.
2      Written by:
3      Date written:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9      // Local Declarations
10     char*  pDays[7];
11     char** pLast;
12
13     // Statements
14     pDays[0] = "Sunday";
15     pDays[1] = "Monday";
16     pDays[2] = "Tuesday";
17     pDays[3] = "Wednesday";
18     pDays[4] = "Thursday";
19     pDays[5] = "Friday";
20     pDays[6] = "Saturday";
21
22     printf("The days of the week\n");
23     pLast = pDays + 6;
24     for (char** pWalker = pDays;
25         pWalker <= pLast;
26         pWalker++)
27         printf("%s\n", *pWalker);
28     return 0;
29 } // main

```

## C Programming & Data Structures



### Array applications:

There are two array applications: frequency arrays and histograms.

A frequency array shows the number of elements with an identical value found in a series of numbers.

A histogram is a pictorial representation of a frequency array. Instead of printing the values of the elements to show the frequency of each number, we print a histogram in the form of a bar chart.

## POINTERS

Pointer is a user defined data type which creates special types of variables which can hold the address of primitive data type like char, int, float, double or user defined data type like function, pointer etc. or derived data type like array, structure, union, enum. Pointers are widely used in programming; pointer stores a **reference** to another value. The variable the pointer refers to is sometimes known as its "pointee". The following fig: shows two variables: num and numPtr. The simple variable num contains the value 42 in the usual way. The variable numPtr is a pointer which contains a reference to the variable num. The numPtr variable is the pointer and num is its pointee. What is stored inside of numPtr? Its value is not an int. Its value is a reference to an int.

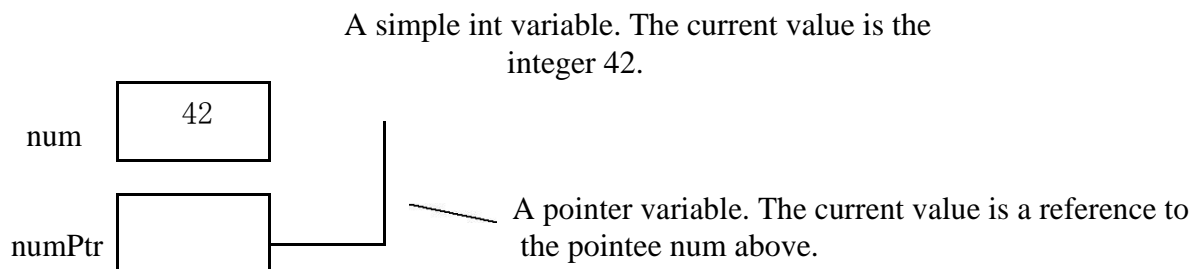


Fig:1

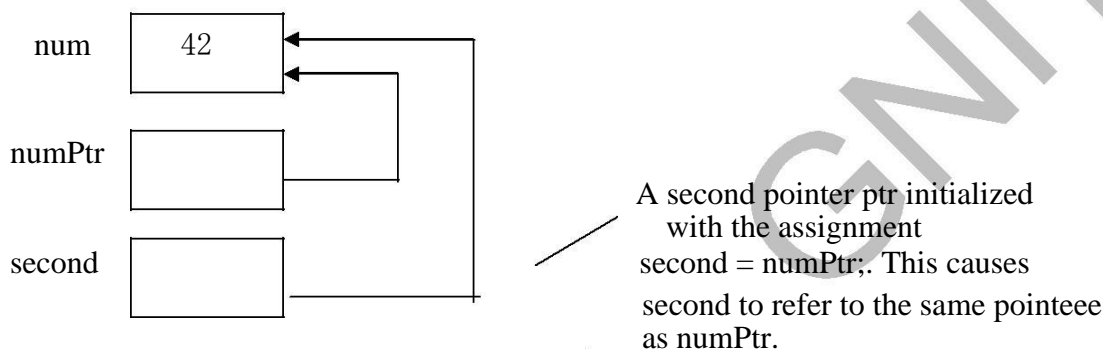
## C Programming & Data Structures

### Pointer Dereference

The "dereference" operation follows a pointer's reference to get the value of its pointee. The value of the dereference of numPtr above is 42. The only restriction is that the pointer must have a pointee for the dereference to access.

### Pointer Assignment

The assignment operation (=) between two pointers makes them point to the same pointee. The example below adds a second pointer, second, assigned with the statement `second = numPtr;`. The result is that second points to the same pointee as numPtr. In the drawing, this means that the second and numPtr boxes both contain arrows pointing to num. Assignment between pointers does not change or even touch the pointers. It just changes which pointers a pointer refers to.



After assignment, the `==` test comparing the two pointers will return true. For example `(second==numPtr)` above is true.

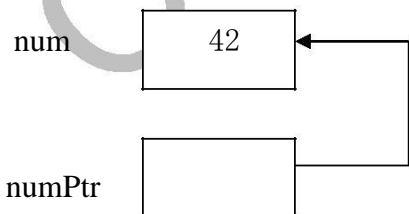
### Pointer Variables

Pointer variables are declared just like any other variable. The declaration gives the type and name of the new variable and reserves memory to hold its value. The declaration does not assign a pointee for the pointer

```
int* numPtr; // Declare the int* (pointer to int) variable "numPtr". //
             This allocates space for the pointer, but not the pointee.
```

### The & Operator — Reference To

There are several ways to compute a reference to a pointee suitable for storing in a pointer. The simplest way is the & operator. The & operator can go to the left of any variable, and it computes a reference to that variable. The code below uses a pointer and an & to produce the earlier num/numPtr example.



```
void NumPtrExample()
{ int num;
```

## C Programming & Data Structures

```
int* numPtr;
num = 42;
numPtr = &num; // Compute a reference to "num", and store it in
numPtr // At this point, memory looks like drawing above
}
```

### Example:

```
void PointerTest() {
// allocate three integers and two
pointers int a = 1;
int b = 2;
int c = 3;
int* p;
int* q;
p = &a; // set p to refer to a
q = &b; // set q to refer to b
c = *p; // retrieve p's pointee value (1) and put it in c
p = q; // change p to share with q (p's pointee is now b)
*p = 13; // dereference p to set its pointee (b) to 13 (*q is now 13)
}
```

### Pointer arithmetic

Pointers can be added and subtracted. Addition and subtraction are mainly for moving forward and backward in an array.

Operator	Result
++	Goes to the next memory location that the pointer is pointing to.
--	Goes to the previous memory location that the pointer is pointing to.
-= or -	Subtracts value from pointer.
+= or +	Adding to the pointer

### Pointers and Arrays:

Just like other variables, we have to declare pointers before we can use them. Pointer declarations look much like other declarations. When pointers are declared, the keyword at the beginning (c int, char and so on) declares the type of variable that the pointer will point to. The pointer itself is not of that type, it is of type pointer to that type. A given pointer only points to one particular type, not to all possible types. Here's the declaration of an array and a pointer:

```
int ar[5], *ip;
```

The \* in front of ip in the declaration shows that it is a pointer, not an ordinary variable. It is of type pointer to int, and can only be used to refer to variables of type int. It's still uninitialized; it has to be made to point to something.

```
int ar[5], *ip;
```

## C Programming & Data Structures

```
ip = &ar[3];
```

In the example, the pointer is made to point to the member of the array ar whose index is 3, i.e. the fourth member. we can assign values to pointers just like ordinary variables; the difference is simply in what the value means.

Example:

Array and pointer arithmetic

Sample Code

```
1. #include <stdio.h>
2. int main()
3. {
4.     int ArrayA[3]={ 1,2,3};
5.     int *ptr;
6.     ptr=ArrayA;
7.     printf("address: %p - array value:%d n",ptr,*ptr);
8.     ptr++;
9.     printf("address: %p - array value:%d n",ptr,*ptr);
10.    return 0;
11. }
```

Description : In line 1 we are declaring „ArrayA“ integer array variable initialized to numbers „1,2,3“, in line 2, the pointer variable ptr is declared. In line 3, the address of variable „ArrayA“ is assigned to variable ptr. In line 5 ptr is incremented by 1.

Note: & notation should not be used with arrays because array's identifier is pointer to the first element of the array.

### Character pointer

The array declaration "char a[6];" requests that space for six characters be set aside, to be known by the name "a." That is, there is a location named "a" at which six characters can sit. The pointer declaration "char \*p;" on the other hand, requests a place which holds a pointer. The pointer is to be known by the name "p," and can point to any char (or contiguous array of chars) anywhere.

The statements

```
char a[] = "hello";
char *p = "world";
```

would result in data structures which could be represented like this:

```
a: | h | e | l | l | o |
p: | *=====> | w | o | r | l | d | \0 |
```

It is important to realize that a reference like x[3] generates different code depending on whether x is an array or a pointer. Given the declarations above, when the compiler sees the expression a[3], it emits code to start at the location "a," move three past it, and fetch the character there. When it sees the expression p[3], it emits code to start at the location "p," fetch the pointer value there, add three to the pointer, and finally fetch the character pointed to. In the example above, both a[3] and p[3] happen to be the character 'l', but the compiler gets there differently.

Example:



## C Programming & Data Structures

```

1. #include <stdio.h>
2. int main()
3. {
4.     char a='b';
5.     char *ptr;
6.     printf("%cn",a);
7.     ptr=&a;
8.     printf("%pn",ptr);
9.     *ptr='d';
10.    printf("%cn",a);
11.    return 0;
12. }

```

Output :

```

b
001423
d

```

Description: In line 1 we are declaring char variable called a; it is initialized to character „b“, in line 2, the pointer variable ptr is declared. In line 4, the address of variable a is assigned to variable ptr. In line 6 value stored at the memory address that ptr points to is changed to „d“

Note: & notation means address-of operand in this case &a means „address-of a“.

### Pointers and 2 Dimensional Arrays

A 2D array in C is treated as a 1D array whose elements are 1D arrays (the rows). Example:

A 4x3 array of T (where "T" is some data type) may be declared by: "T mat[4][3]", and described by the following scheme:

```

mat == mat[0] ---> | a00 | a01 | a02 |
mat[1] ---> | a10 | a11 | a12 |
mat[2] ---> | a20 | a21 | a22 |
mat[3] ---> | a30 | a31 | a32 |

```

The array elements are stored in memory row after row, so the array equation for element "mat[m][n]" of type T is:

```

address(mat[i][j]) = address(mat[0][0]) + (i * n + j) * size(T)
address(mat[i][j]) = address(mat[0][0]) + i * n * size(T) + j * size(T)
address(mat[i][j]) = address(mat[0][0]) + i * size(row of T) + j * size(T)

```



## C Programming & Data Structures

### Example:

```
#include <stdio.h>
int main(void){
    char board[3][3] = {
        {'1','2','3'},
        {'4','5','6'},
        {'7','8','9'}
    };

    int i;
    for(i = 0; i < 9; i++)
        printf(" board: %c\n", *(*board +
i)); return 0;
}
```

Output:

```
board: 1
board: 2
board: 3
board: 4
board: 5
board: 6
board: 7
board: 8
board: 9
```

### Pointers and functions

Pointers can be used with functions. The main use of pointers is „call by reference“ functions. Call by reference function is a type of function that has pointer/s (reference) as parameters to that function. All calculation of that function will be directly performed on referred variables.

Example

```
1. #include <stdio.h>
2. void DoubleIt(int *num)
3. {
4.     *num*=2;
5. }
6. int main()
7. {
8.     int number=2;
9.     DoubleIt(&number);
10.    printf("%d",number);
11. return 0;
12. }
```

Output

4

## C Programming & Data Structures

---

Description : in line 1 we are declaring „DoubleIt“ function, in line 4, the variable number is declared and initialized to 2. In line 5, the function DoubleIt is called.

### Pointers and Structures

Pointers and structures is broad topic and it can be very complex However pointers and structures are great combinations; linked lists, stacks, queues and etc are all developed using pointers and structures in advanced systems.

Example:

```
1. #include <stdio.h>
2.
3. struct details {
4.     int num;
5. };
6.
7. int main()
8. {
9.
10. struct details MainDetails;
11. struct details *structptr;
12. structptr=&MainDetails;
13. structptr->num=20;
14. printf("n%d",MainDetails.num);
15.
16.
17. return 0;
18. }
```

Output

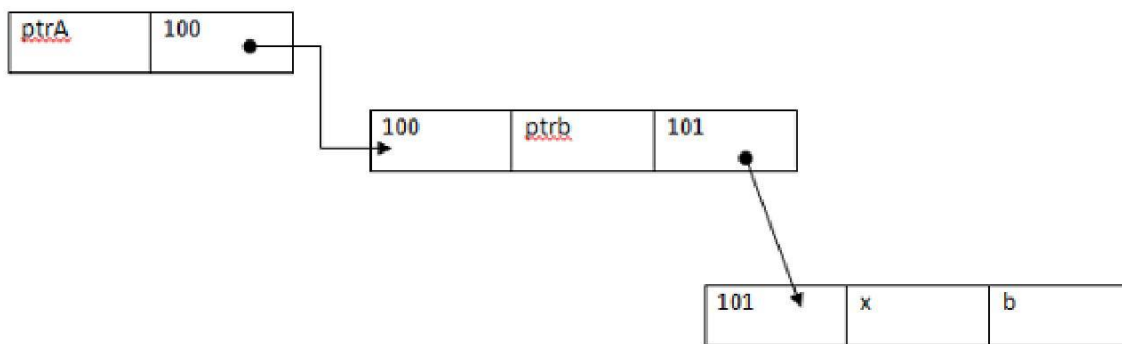
20

Description: in line 1-3 we are declaring „details“ structure, in line 4, the variable Maindetails is declared.in line 6, pointer is set to point to MainDetails. In line 7, 20 is assigned to MainDetails.num through structptr->num.

### Pointer to Pointer

Pointers can point to other pointers; there is no limit whatsoever on how many „pointer to pointer“ links you can have in your program. It is entirely up to you and your programming skills to decide how far you can go before you get confused with the links. Here we will only look at simple pointer to pointer link. Pointing to pointer can be done exactly in the same way as normal pointer. Diagram below can help you understand pointer to pointer relationship.

## C Programming & Data Structures



### Example

```

Char *ptr A;
Char x="b";
Char *ptr b;
ptr b=&x;
ptr A=&ptr b;
  
```

Ptr A gives 100, \*ptr A gives 101, ptr b is 101, \*ptr b gives b, \*\*ptr A gives b

comment: \*\*ptr A means „value stored at memory location of value stored in memory location stored at Ptr A“

### Pointers to Functions

A pointer to a function points to the address of the executable code of the function. You can use pointers to call functions and to pass functions as arguments to other functions. You cannot perform pointer arithmetic on pointers to functions. The type of a pointer to a function is based on both the return type and parameter types of the function.

A declaration of a pointer to a function must have the pointer name in parentheses. The function call operator () has a higher precedence than the dereference operator \*. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type. For example:

```

int *f(int a);    /* function f returning an int* */
int (*g)(int a); /* pointer g to a function returning an int */
char (*h)(int, int) /* h is a function
                    that takes two integer parameters and returns char */
  
```

In the first declaration, f is interpreted as a function that takes an int as argument, and returns a pointer to an int. In the second declaration, g is interpreted as a pointer to a function that takes an int argument and that returns an int.

Once we have got the pointer, we can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression. we can call the function using one of two forms:

```

(*func)(1,2);
/* or */
func(1,2);
  
```

The second form has been newly blessed by the Standard. Here's a simple example

```

#include <stdio.h>
#include <stdlib.h>
void func(int);
  
```

## C Programming & Data Structures

```
main(){
    void
    (*fp)(int); fp =
    func; (*fp)(1);
    fp(2);
    exit(EXIT_SUCCESS);
}
```

```
void
func(int arg){
    printf("%d\n", arg);
}
```

If we like writing finite state machines, we might like to know that we can have an array of pointers to functions, with declaration and use like this:

Example

```
void (*fparr[])(int, float) = {
    /* initializers */
};
/* then call one */
fparr[5](1, 3.4);
```

### Pointers to Void

A void pointer is a C convention for “a raw address.” The compiler has no idea what type of object a void Pointer “really points to.” If we write `int *ip`; `ip` points to an `int`. If we write `void *p`; `p` doesn’t point to a void! In C, any time we need a void pointer, we can use another pointer type. For example, if you have a `char*`, we can pass it to a function that expects a `void*`. we don’t even need to cast it. In C, we can use a `void*` any time you need any kind of pointer, without casting. A void pointer is used for working with raw memory or for passing a pointer to an unspecified type. Some C code operates on raw memory. When C was first invented, character pointers (`char *`) were used for that. Then people started getting confused about when a character pointer was a string, when it was a character array, and when it was raw memory.

Example: `#include <stdio.h>`

```
void use_int(void *);
void use_float(void *);
void greeting(void (*)(void *), void *);
int main(void) {
    char ans;
    int i_age = 22;
    float f_age =
    22.0; void *p;
    printf("Use int (i) or float (f)?
    "); scanf("%c", &ans);
    if (ans == 'i') {
        p = &i_age;
        greeting(use_int, p);
    }
```

## C Programming & Data Structures

```

else {
    p = &f_age;
    greeting(use_float, p);
}
return 0;
}

void greeting(void (*fp)(void *), void *q)
{ fp(q);
}

void use_int(void *r)
{ int a;
  a = * (int *) r;
  printf("As an integer, you are %d years old.\n", a);
}

void use_float(void *s)
{ float *b;
  b = (float *) s;
  printf("As a float, you are %f years old.\n", *b);
}

```

Although this requires us to cast the void pointer into the appropriate type in the relevant subroutine (use\_int or use\_float), the flexibility here appears in the greeting routine, which can now handle in principle a function with any type of argument.

### Compatibility of pointers

Two pointer types with the same type qualifiers are compatible if they point to objects of compatible types. The composite type for two compatible pointer types is the similarly qualified pointer to the composite type

The following example shows compatible declarations for the assignment

```

operation: float subtotal;
float * sub_ptr;
/* ... */
sub_ptr = &subtotal;
printf("The subtotal is %f\n", *sub_ptr);

```

The next example shows incompatible declarations for the assignment

```

operation: double league;
int * minor;
/* ... */
minor = &league; /* error */

```

### Array of Pointers

A pointer is a variable that contains the memory location of another variable. The values you assign to the pointers are memory addresses of other variables (or other pointers). A running program gets a certain space in the main memory.

Syntax of declaring a pointer:

```
data_type_name * variable name
```

Specify the data type of data stored in the location, which is to be identified by the pointer. The asterisk tells the compiler that we are creating a pointer variable. Then specify the name of variable.

## C Programming & Data Structures

### Example:

```
#include <stdio.h>
#include
<conio.h> main() {
    clrscr();
    int *array[3];
    int x = 10, y = 20, z =
    30; int i;
    array[0] = &x;
    array[1] = &y;
    array[2] = &z;
    for (i=0; i< 3; i++) {
        printf("The value of %d= %d ,address is %u\t\n", i,
            *(array[i]), array[i]);
    }
    getch();
    return 0;
}
```

### Output:

```
The value of 0 = 10, address is 65518
The value of 1 = 20, address is 65516
The value of 2 = 30, address is 65514
```

## Strings

### Definition:

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the *null character*, which is simply the character with the value 0. (The null character has no relation except in name to the *null pointer*. In the ASCII character set, the null character is named NULL.) The null or string-terminating character is represented by another character escape sequence \0. For example, we can declare and define an array of characters, and initialize it with a string constant:

```
char string[] = "Hello, world!";
```

C also permits us to initialize a character array without specifying the number of elements.

```
char string[] = {'G','O','O','D','\0'};
```

### Reading strings:

The familiar input function scanf can be used with %s format to read in a string of characters. char address[15];

```
scanf("%s",address);
```

### Writing strings to screen:

```
printf("%s",address);
```

Another more convenient method of reading string of text containing white spaces is to use the library function gets available in the <stdio.h> header file.

## C Programming & Data Structures

```
Ex: char line[10];
    gets(line);
```

The above set of statements read a string into the character array line.

Another more convenient method of printing string values is to use the library function puts() available in <stdio.h> header file.

```
Ex: char line[10];
    gets(line);
    puts(line);
```

The above set of statements print a string on the screen.

### Library functions for strings

Function	Library	Description
strcpy(s1, s2)	string.h	Copies the value of s2 into s1
strncpy(s1, s2, n)	string.h	Copies n characters of s2 into s1. Does not add a null.
strcat(s1, s2)	string.h	Appends s2 to the end of s1
strncat(s1, s2, n)	string.h	Appends n characters of s2 onto the end of s1
strcmp(s1, s2)	string.h	Compares s1 and s2 alphabetically; returns a negative value if s1 should be first, a zero if they are equal, or a positive value if s2 should be first
strncmp(s1, s2, n)	string.h	Compares the first n characters of s1 and s2 in the same manner as strcmp
strlen(s1)	string.h	Returns the number of characters in s1 not counting the null

Since C never lets us assign entire arrays, we use the strcpy function to copy one string to another: #include <string.h>

```
char string1[] = "Hello,
world!"; char string2[20];
strcpy(string2, string1);
```

The above code copies the contents of string1 to string2.

The standard library's strcmp function compares two strings, and returns 0 if they are identical, or a negative number if the first string is alphabetically "less than" the second string, or a positive number if the first string is "greater." Here is an example:

```
char string3[] = "this is";
char string4[] = "a test";

if(strcmp(string3, string4) == 0)
    printf("strings are equal\n");
```

## C Programming & Data Structures

---

else printf("strings are different\n"); This code fragment will print ``strings are different".

Another standard library function is strcat, which concatenates strings. It does *not* concatenate two strings together and give you a third, new string; what it really does is append one string onto the end of another. Here's an example:

```
char string5[20] = "Hello,
"; char string6[] = "world!";
printf("%s\n", string5);
strcat(string5, string6);
printf("%s\n", string5);
```

The first call to printf prints ``Hello, ", and the second one prints ``Hello, world!", indicating that the contents of string6 have been tacked on to the end of string5. Notice that we declared string5 with extra space, to make room for the appended characters.

The length of the string can be found by using the function strlen() function. char string7[] = "abc";

```
int len = strlen(string7);
printf("%d\n", len);
```

### String/Data conversion:

The C standard includes two sets of memory formatting functions, one for ASCII characters and one for wide characters, that uses these same formatting functions. Rather than reading and writing files , however they read and write strings in memory.

### String to data conversion:

The string scan function is called sscanf. This function scans a string as though the data were coming from a file.

**Syntax:** int sscanf(char \*str,const char \*frmt\_str,...);

The first parameter specifies the string holding the data to be scanned. The .... Indicates that a variable number of pointers identify the fields into which the formatted data are to be placed.

### Data to String conversion:

The string print function,sprintf, follows the rules of fprintf. Rather than sending the data to a file, it simply writes them to a string.

**Syntax:** int sprintf(char \*out\_string,const char \*format\_string,...);

The first parameter is a pointer to a string that will contain the formatted output. The format string is the same as printf and follows all of the same rules. The ... contains the fields that correspond to the format codes in the format string.



# C Programming & Data Structures

## Unit V

### Structures

A structure is a collection of one or more variables, possibly of different data types, grouped together under a single name for convenient handling. A structure is a convenient way of grouping several pieces of related information together.

**Structure** is user defined data type which is used to store heterogeneous data under unique name. Keyword 'struct' is used to declare structure.

The variables which are declared inside the structure are called as 'members of structure'.

A structure can be defined as a new named type using typedef, thus extending the number of available types.

#### Defining a Structure(Using struct)

A structure type is usually defined near to the start of a file. Structures must be defined first and then declared. Structure definition just describes the format. A structure type is usually defined near to the start of a file using either by a struct keyword or by typedef statement.

To define a structure, We use **struct** keyword

#### Syntax of structure definition:

```
struct tag_name
{
    data_type member_1;
    data_type member_2;
    data_type member_n;
};
```

#### Example: struct book

```
{
    char name[20];
    char author[20];
    int pages;
    float price;
};
```

This defines a book structure with the specified members.

#### Defining a Structure(using typedef)

typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

Example structure definition.

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

## C Programming & Data Structures

---

```
    int year;
} student;
```

This defines a new type student variables of type student

### Declaring Structure Variables:

The first way is to declare a structure followed by structure definition like this

```
: struct structure_name
{
    structure _member;
    ....
}instance_1,instance_2,..instance_n;
```

In the second way, we can declare the structure instance at a different location in our source code after structure definition(in the main function).

Here is structure declaration syntax :

```
    struct struct_name instance_1,instance_2
instance_n; Example: struct student s;
```

### Shorthand structure with *typedef* keyword

To make the source code more concise, we can use *typedef* keyword to create a synonym for a structure. This is an example of using *typedef* keyword to define address structure, so when we want to create an instance of it we can omit the keyword *struct*

```
typedef struct
{
    unsigned int house_number;
    char street_name[50];
    int zip_code;
    char country[50];
} address;

address billing_addr;
address shipping_addr;
```

### Accessing Members of a Structure

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. In the above, member name of structure „s“ will behave just like a normal array of char, however we refer to it by the name

```
s.name
```

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

```
st_ptr -> name
```

## C Programming & Data Structures

### Initialization of structures

**C99** allow the initializer for an automatic member variable of a structure type to be a constant or non-constant expression.

There are two ways to specify initializers for structures and unions:

- **With C89-style** initializers, structure members must be initialized in the order declared, and only the first member of a union can be initialized.
- **C99 feature** C99 feature Using *designated* initializers, which allows you to *name* members to be initialized, structure members can be initialized in any order, and any (single) member of a union can be initialized.

### Designated initializers for aggregate types (C only) a C99 feature

*Designated initializers*, are supported for aggregate types, including arrays, structures, and unions. A designated initializer, or *designator*, points out a particular element to be initialized. A *designator list* is a comma-separated list of one or more designators. A designator list followed by an equal sign constitutes a *designation*.

Designated initializers allow for the following flexibility:

- Elements within an aggregate can be initialized in any order.
- The initializer list can omit elements that are declared anywhere in the aggregate, rather than only at the end. Elements that are omitted are initialized as if they are static objects: arithmetic types are initialized to 0; pointers are initialized to NULL.
- Where inconsistent or incomplete bracketing of initializers for multi-dimensional arrays or nested aggregates may be difficult to understand, designators can more clearly identify the element or member to be initialized.

In the following example, the designator is `.any_member` and the designated initializer is `.any_member = 13`:

The following example shows how the second and third members `b` and `c` of structure variable `klm` are initialized with designated initializers:

```
struct xyz
{
    int a;
    int b;
    int c;
} klm = { .a = 99, .c = 100 };
```

We can also use designators to represent members of nested structures. For example:

```
struct a
{
    struct b
    {
        int c;
```

## C Programming & Data Structures

```

    int d;
} e;
float f;
} g = {.e.c = 3 };

```

Initializes member c of structure variable e, which is a member of structure variable g, to the value of 3.

### A completely initialized

```

structure: struct address {
    int street_no;
    char street_name[20];
    char city[10];
    char dist[10];
    char
    postal_code[10]; };
    static struct address perm_address =
        { 3, "asdf.", "hyderabad", "RRdist", "500000"};

```

### Example Program:

```

#include <stdio.h>
#include <conio.h>
struct student
{
    int id;
    char *name;
    float percentage;
} student1, student2,
student3; int main()
{
    struct student st;
    student1.id=1;
    student2.name = "xxxx";
    student3.percentage = 90.5;
    printf(" Id is: %d \n", student1.id); printf("
    Name is: %s \n", student2.name);
    printf(" Percentage is: %f \n",
    student3.percentage); getch();
    return 0;
}

```

### Output will be displayed as:

```

    Id is : 1
    Name is : xxxx
    Percentage is : 90.500000

```

## C Programming & Data Structures

---

### Structures and functions

A structure can be passed as a function argument just like any other variable. This raises a few practical issues.

A structure can be passed to a function in 3 ways.

- (i) Passing individual members of the structure to a function.
- (ii) Passing a copy of the structure variable( Call by value).
- (iii) Passing a structure by passing its pointer to the structure.

When we wish to modify the value of members of the structure, we must pass a pointer to that structure. This is just like passing a pointer to an int type argument whose value we wish to change.

If we are only interested in one member of a structure, it is probably simpler to just pass that member. This will make for a simpler function, which is easier to re-use. Of course if we wish to change the value of that member, we should pass a pointer to it.

When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

### Nested structures(Structures within Structures) :

Structures can be used as structures within structures. It is also called as 'nesting of structures'.

#### Syntax:

```
struct structure_nm
{
    <data-type> element 1;
    <data-type> element 2;
    -----
    -----
    <data-type> element n;

    struct structure_nm
    {
        <data-type> element 1;
        <data-type> element 2;
        -----
        -----
        <data-type> element
        n; }inner_struct_var;
    }outer_struct_var;
```

#### Example :

## C Programming & Data Structures

---

```

struct stud_Res
{
    int rno;
    char nm[50];
    char std[10];

    struct stud_subj
    {
        char subjnm[30];
        int marks;
    }subj;
}result;

```

In above example, the structure stud\_Res consists of stud\_subj which itself is a structure with two members. Structure stud\_Res is called as 'outer structure' while stud\_subj is called as 'inner structure.'

The members which are inside the inner structure can be accessed as follow :

```

result.subj.subjnm
result.subj.marks

```

### Program:

```

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

```

```

struct stud_Res
{
    int rno; char
    std[10];
    struct stud_Marks
    {
        char subj_nm[30];
        int subj_mark;
    }marks;
}result;

void main()
{
    clrscr();
    printf("\n\t Enter Roll Number : ");
    scanf("%d",&result.rno);
    printf("\n\t Enter Standard : ");
    scanf("%s",result.std);
    printf("\n\t Enter Subject Code : ");
    scanf("%s",result.marks.subj_nm); printf("\n\t
Enter Marks : ");
    scanf("%d",&result.marks.subj_mark);
    printf("\n\n\t Roll Number : %d",result.rno);

```

## C Programming & Data Structures

---

```
printf("\n\n\t Standard : %s",result.std);
printf("\nSubject Code :
%s",result.marks.subj_nm); printf("\n\n\t Marks :
%d",result.marks.subj_mark); getch();
}
```

### **Output :**

```
Enter Roll Number : 1
Enter Standard : BTECH-I
Enter Subject code : SUB001
Enter Marks :63
```

```
Roll Number : 1
Standard : BTECH-I
Subject code : SUB001
Marks :63
```

### **Arrays within Structures:**

Sometimes, it is necessary to use structure members with array.

### **Program :**

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

struct result
{
    int rno, mrks[5];
    char nm;
}res;

void main()
{
    int i,total;
    clrscr();
    total = 0;
    printf("\n\t Enter Roll Number :
"); scanf("%d",&res.rno);
    printf("\n\t Enter Marks of 3 Subjects :
"); for(i=0;i<3;i++)
    {
        scanf("%d",&res.mrks[i]);
        total = total + res.mrks[i];
    }
    printf("\n\n\t Roll Number :
%d",res.rno); printf("\n\n\t Marks are :");
    for(i=0;i<3;i++)
    {
```

## C Programming & Data Structures

---

```

        printf(" %d",res.mrks[i]);
    }
    printf("\n\n\t Total is :
    %d",total); getch();
}

```

### **Output :**

```

Enter Roll Number : 1
Enter Marks of 3 Subjects : 63 66 68
Roll Number :1
Marks are : 63 66 68
Total is : 197

```

### **Arrays of structures:**

An array of structures for the example is declared in the usual way: 100

```

books struct book
{
    char  name[20];
    char  author[20];
    int  pages;
    float price;
} struct student[100];

```

The members of the structures in the array are then accessed by statements such as the following:

The value of a member of a structure in an array can be assigned to another variable, or the value of a variable can be assigned to a member. For example

```
book [3].pages = 794;
```

Like all other arrays in C, struct arrays start their numbering at zero.

The following code assigns the value of the „price“ member of the fourth element of „book“ to the variable „p“:

```
p = book[3].price;
```

Finally, the following example assigns the values of all the members of the second element of book, namely book[1], to the third element, so book[2] takes the overall value of book[1].

```
book[2] = book[1];
```

We can create structures with array for ease of operations in case of getting multiple same fields.

### **Program :**

```

#include <stdio.h>
#include <conio.h>
struct emp_info
{
    int emp_id;
    char nm[50];
}emp[2];

void main()
{

```



## C Programming & Data Structures

---

```
int i;
clrscr();
for(i=0;i<2;i++)
{
    printf("\n\n\t Enter Employee ID : ");
    scanf("%d",&emp[i].emp_id);
    printf("\n\n\t Employee Name : ");
    scanf("%s",emp[i].nm);
}
for(i=0;i<2;i++)
{
    printf("\n\t Employee ID : %d",emp[i].emp_id);
    printf("\n\t Employee Name : %s",emp[i].nm);
}
getch();
}
```

### **Output :**

```
Enter Employee ID : 1
Employee Name : ABC
Enter Employee ID : 2
Employee Name : XYZ
```

```
Enter Employee ID : 1
Employee Name : ABC
Enter Employee ID : 2
Employee Name : XYZ
```

### **Pointers to structures:**

A structure can contain a pointer to another structure of its own type, or even to itself. This is because a pointer to a structure is not itself a structure, but merely a variable that holds the address of a structure. Pointers to structures are quite invaluable, in fact, for building data structures such as linked lists and trees.

A pointer to a structure type variable is declared by a statement such as the following:

```
struct student *st_ptr;
```

The `st_ptr` is a pointer to a variable of type `struct student`. This pointer can be assigned to any other pointer of the same type, and can be used to access the members of its structure. According to the rules we have outlined so far, this would have to be done like so:

```
struct student s1;
st_ptr = &s1;
(*st_ptr).rollno = 23;
```

## C Programming & Data Structures

---

This code example says, in effect, "Let the member rollno of the structure pointed to by st\_ptr take the value 23." The use of parentheses to avoid confusion about the precedence of the \* and . operators.

There is a better way to write the above code, however, using a new operator: ->. This is an arrow made out of a minus sign and a greater than symbol, and it is used as follows:

```
st_ptr->rollno = 23;
```

The -> enables us to access the members of a structure directly via its pointer. This statement means the same as the last line of the previous code example, but is considerably clearer. The -> operator will come in very handy when manipulating complex data structures

### Pointers in Structures

A structure can contain pointers as structure members and we can create a pointer to a structure as follows:

```
struct invoice
{
    char* code;
    char date[20];
};

struct address billing_addr; struct
address *pa = &billing_addr;
```

### Other Applications of Structures

As we have seen, a structure is a good way of storing related data together. It is also a good way of representing certain types of information. Complex numbers in mathematics inhabit a two dimensional plane (stretching in real and imaginary directions). These could easily be represented here by

```
typedef struct
{
    double real;
    double imag;
} complex;
```

doubles have been used for each field because their range is greater than floats and because the majority of mathematical library functions deal with doubles by default.

In a similar way, structures could be used to hold the locations of points in multi-dimensional space. Mathematicians and engineers might see a storage efficient implementation for sparse arrays here.

Apart from holding data, structures can be used as members of other structures. Arrays of structures are possible, and are a good way of storing lists of data with regular fields, such as databases.

Another possibility is a structure whose fields include pointers to its own type. These can be used to build chains (programmers call these linked lists), trees or other connected structures. These are rather daunting to the new programmer, so we won't deal with them here.

---

## C Programming & Data Structures

### UNIONS

Union is user defined data type used to stored data under unique variable name at single memory location. Union is similar to that of structure. Syntax of union is similar to structure.

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

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

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

#### Syntax:

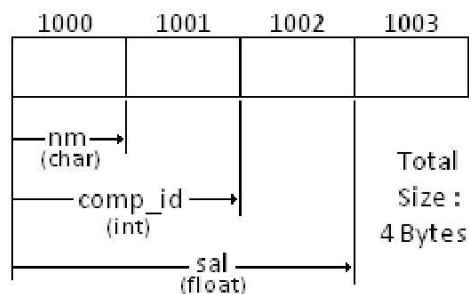
```
union union_name
{
    <data-type> element 1;
    <data-type> element 2;
    <data-type> element 3;
}union_variable;
```

#### Example:

```
union techno
{
    int comp_id;
    char nm;
    float sal;
}tch;
```

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

#### MEMORY ALLOCATION :



To access **union members**, we can use the following

syntax. tch.comp\_id

tch.nm

tch.sal

#### Program:

```
#include <stdio.h>
#include <conio.h>
union techno
{
    int id;
    char nm[50];
```

## C Programming & Data Structures

```

}tch;
void main()
{
    clrscr();
    printf("\n\t Enter developer id : ");
    scanf("%d", &tch.id);
    printf("\n\t Enter developer name : ");
    scanf("%s", tch.nm);
    printf("\n\n Developer ID : %d", tch.id); //Garbage
    printf("\n\n Developed By : %s", tch.nm);
    getch();
}

```

Output :

```

Enter developer id : 101
Enter developer name : technowell
Developer ID : 25972
Developed By : technowell_

```

### Difference between Structure and Union:

Structure	Union
<b>i. Access Members</b>	
We can access all the members of structure at	Only one member of union can be accessed at anytime. anytime.
<b>ii. Memory Allocation</b>	
Memory is allocated for all variables.	Allocates memory for variable which variable require more memory.
<b>iii. Initialization</b>	
All members of structure can be initialized	Only the first member of a union can be initialized.
<b>iv. Keyword</b>	
'struct' keyword is used to declare structure.	'union' keyword is used to declare union.
<b>v. Syntax</b>	
<b>struct</b> struct_name { structure element 1; structure element 2; ----- ----- structure element n; }struct_var_nm;	<b>union</b> union_name { union element 1; union element 2; ----- ----- union element n; }union_var_nm;
<b>vi. Example</b>	
<b>struct</b> item_mst { int rno; char nm[50]; }it;	<b>union</b> item_mst { int rno; char nm[50]; }it;

## C Programming & Data Structures

---

### Enumerations

An *enumeration* is a data type consisting of a set of named values that represent integral constants, known as *enumeration constants*.

An enumeration also referred to as an *enumerated type* because we must list (enumerate) each of the values in creating a name for each of them. In addition to providing a way of defining and grouping sets of integral constants, enumerations are useful for variables that have a small number of possible values.

ENUM is closely related to the `#define` preprocessor.

It allows you to define a list of aliases which represent integer numbers. For example if you find yourself coding something like:

```
#define MON 1
#define TUE 2
#define WED 3
```

You could use enum as below.

```
enum week { Mon=1, Tue, Wed, Thu, Fri Sat, Sun } days;
```

or

```
enum escapes { BELL = '\a', BACKSPACE = '\b', HTAB = '\t',
              RETURN = '\r', NEWLINE = '\n', VTAB = '\v' };
```

or

```
enum boolean { FALSE = 0, TRUE };
```

An advantage of enum over `#define` is that it has scope. This means that the variable (just like any other) is only visible within the block it was declared within.

We can declare an enumeration type separately from the definition of variables of that type. We can define an enumeration data type and all variables that have that type in one statement

### Enumeration type definition

An enumeration type definition contains the `enum` keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. A comma separates each enumerator in the enumerator list. C99 allows a trailing comma between the last enumerator and the closing brace.

Enumeration definition syntax

```
enum tag_identifier { enumerators list };
```

The *tag\_identifier* gives a name to the enumeration type. If we do not provide a tag name, we must put all variable definitions that refer to the enumeration type within the declaration of the type. Similarly, we cannot use a type qualifier with an enumeration definition; type qualifiers placed in front of the `enum` keyword can only apply to variables that are declared within the type definition.

### Enumeration members

The list of enumeration members, or *enumerators*, provides the data type with a set of values.

Enumeration member declaration syntax

```
Identifier = enumeration constant
```

In C, an *enumeration constant* is of type `int`. If a constant expression is used as an initializer, the value of the expression cannot exceed the range of `int` (that is, `INT_MIN` to `INT_MAX` as defined in the header `limits.h`).

## C Programming & Data Structures

In C++, each enumeration constant has a value that can be promoted to a signed or unsigned integer value and a distinct type that does not have to be integral. You can use an enumeration constant anywhere an integer constant is allowed, or anywhere a value of the enumeration type is allowed.

The value of a constant is determined in the following way:

- ⤴ An equal sign (=) and a constant expression after the enumeration constant gives an explicit value to the constant. The identifier represents the value of the constant expression.
- ⤴ If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).
- ⤴ Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

The following data type declarations list oats, wheat, barley, corn, and rice as enumeration constants. The number under each constant shows the integer value.

```
enum grain { oats, wheat, barley, corn, rice
}; /* 0 1 2 3 4 */
```

It is possible to associate the same integer with two different enumeration constants. For example, the following definition is valid. The identifiers suspend and hold have the same integer value.

```
enum status { run, clear=5, suspend, resume, hold=6
}; /* 0 5 6 7 6 */
```

Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, the second declarations of average and poor cause compiler errors:

```
func()
{
    enum score { poor, average, good };
    enum rating { below, average, above
    }; int poor;
}
```

### Enumeration variable declarations

We must declare the enumeration data type before we can define a variable having that type. Enumeration variable declaration syntax

```
Storage_class type_qualifier enum tag_identifier;
```

The *tag\_identifier* indicates the previously-defined data type of the enumeration.

### Enumeration type and variable definitions in a single statement

We can define a type and a variable in one statement by using a declarator and an optional initializer after the type definition. To specify a storage class specifier for the variable, we must put the storage class specifier at the beginning of the declaration. For example:

```
register enum score { poor=1, average, good } rating = good;
```

Either of these examples is equivalent to the following two declarations:

```
enum score { poor=1, average, good
}; register enum score rating = good;
```

Both examples define the enumeration data type score and the variable rating. rating has the storage class specifier register, the data type enum score, and the initial value good.

Combining a data type definition with the definitions of all variables having that data type lets you leave the data type unnamed. For example:

```
enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday } weekday;
```

## C Programming & Data Structures

This defines the variable `weekday`, which can be assigned any of the specified enumeration constants. However, we can not declare any additional enumeration variables using this set of enumeration constants.

### Example:

```
#include <stdio.h>
int main()
{
    enum Days{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};

    Days TheDay;
    int j = 0;
    printf("Please enter the day of the week (0 to
    6)\n"); scanf("%d",&j);
    TheDay = Days(j);
    if(TheDay == Sunday || TheDay == Saturday)
        printf("Hurray it is the weekend\n");
    else
        printf("Curses still at
        work\n"); return 0;
}
```

### Bit Fields

In addition to declarators for members of a structure or union, a structure declarator can also be a specified number of bits, called a "bit field." Its length is set off from the declarator for the field name by a colon. A bit field is interpreted as an integral type.

**Bit Fields** allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. *e.g.* 1 bit flags can be compacted -- Symbol tables in compilers.
- Reading external file formats -- non-standard file formats could be read in. *E.g.* 9 bit integers.

C lets us do this in a structure definition by putting **:bit length** after the variable.

*i.e.* struct packed\_struct {

```
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int f4:1;
    unsigned int type:4;
    unsigned int funny_int:9;
```

} pack;

Here the packed\_struct contains 6 members: Four 1 bit **flags** f1..f3, a 4 bit type and a 9 bit funny\_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then



## C Programming & Data Structures

some compilers may allow memory overlap for the fields whilst other would store the next field in the next word (see comments on bit fields portability below).

Access members as usual via:

```
pack.type = 7;
```

### NOTE:

- Only  $n$  lower bits will be assigned to an  $n$  bit number. So type cannot take values larger than 15 (4 bits long).
- Bit fields are always converted to integer type for computation.
- You are allowed to mix "normal" types with bit fields.
- The unsigned definition is important - ensures that no bits are used as a flag.

**Syntax:** *struct-declarator:*  
*declarator*

*type-specifier declarator opt : constant-expression*

The *constant-expression* specifies the width of the field in bits.

The *type-specifier* for the declarator must be **unsigned int**, **signed int**, or **int**, and the *constant-expression* must be a nonnegative integer value. If the value is zero, the declaration has no declarator. Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed.

The optional declarator names the bit field. Bit fields can only be declared as part of a structure. The address-of operator (&) cannot be applied to bit-field components.

Unnamed bit fields cannot be referenced, and their contents at run time are unpredictable. They can be used as "dummy" fields, for alignment purposes. An unnamed bit field whose width is specified as 0 guarantees that storage for the member following it in the *struct-declaration-list* begins on an **int** boundary.

Bit fields must also be long enough to contain the bit pattern.

This example defines a two-dimensional array of structures named screen. struct

```
{
    unsigned short icon : 8;
    unsigned short color : 4;
    unsigned short underline :
    1; unsigned short blink : 1;
} screen[25][80];
```

The array contains 2,000 elements. Each element is an individual structure containing four bit-field members: icon, color, underline, and blink. The size of each structure is two bytes.

Bit fields have the same semantics as the integer type. This means a bit field is used in expressions in exactly the same way as a variable of the same base type would be used, regardless of how many bits are in the bit field.

ANSI C standard allows **char** and **long** types (both **signed** and **unsigned**) for bit fields. Unnamed bit fields with base type **long**, **short**, or **char** (**signed** or **unsigned**) force alignment to a boundary appropriate to the base type.



## C Programming & Data Structures

---

### Declaring and Using Bit Fields in Structures

Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. These space-saving structure members are called *bit fields*, and their width in bits can be explicitly declared. Bit fields are used in programs that must force a data structure to correspond to a fixed hardware representation and are unlikely to be portable.

The syntax for declaring a bit field is as follows: *type\_specifier constant\_expression*;

A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant integer expression that indicates the field width in bits, and a semicolon. A bit field declaration may not use either of the type qualifiers, **const** or **volatile**.

#### Example:

Bit fields are allocated within an integer from least-significant to most-significant bit. In the following code

```
#include<stdio.h>
struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;

int main( void );
{
    test.a = 2;
    test.b = 31;
    test.c = 0;
}
```

The bits would be arranged as follows: 00000001 11110010  
ccccccb bbbbaaaa

# C Programming & Data Structures

## Unit VI

### FILE MANAGEMENT IN C

A file is nothing but a place on the disk where a group of related data is stored. Until here we read and write data through console I/O functions *scanf* and *printf*, which always use the keyboard /screen as the target place. If the data is small, it works fine. But many real-life problems involve large volume of data. In such situations the difficulties with above console functions are,

- It becomes cumbersome and time consuming for handling large volume of data.
- When program is terminated / computer is turned off we may lost the entire data.

So we need the concept of files where the data can be stored on the disks and read whenever necessary, without destroying the data.

#### **The basic file operation in 'C'**

- Naming a file
- Opening a file
- Reading data from a file
- Writing data to a file
- Closing file

There are two ways to perform file operations in „C“.

- The low-level I/O and uses UNIX system call.
- The high-level I/O operation and uses „C“ library.

#### **File handling functions in 'C' library**

- fopen( ) → creates a new file for use/open existing file
- fclose( ) → closes file
- getc( ) → read a char from file
- putc( ) → writes a char to file
- fprintf → writes a set of data values to a file
- fscanf → reads a set of data values from a file
- getw( ) → reads an integer from a file
- putw( ) → writes an integer to a file
- fseek( ) → sets the position to a designed point in file
- ftell( ) → sets the position to the beginning of file

#### **Text files**

Text files store character data. A text file can be thought of as a stream of characters that can be processed sequentially. Not only is it processed sequentially, but it can only be processed (logically) in the forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time. Similarly, since text files only process characters, they can only read or write data one character at a time. (Functions are provided that deal with lines of text, but these still essentially process data one character at a time.) A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations.

## C Programming & Data Structures

### Binary files

As far as the operating system is concerned, a binary file is no different to a text file. It is a collection of bytes. In C a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. Since we cannot assume that the binary file contains text, no special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. The interpretation of the file is left to the programmer. C places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be processed sequentially or, depending on the needs of the application, they can (and usually are) processed using random access techniques. In C, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files – they are generally processed using read and write operations simultaneously. For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files. One final note. It is possible, and is sometimes necessary, to open and process a text file as binary data. The operations performed on binary files are similar to text files since both types of files can essentially be considered as streams of bytes. In fact the same functions are used to access files in C. When a file is “opened” it must be designated as text or binary and usually this is the only indication of the type of file being processed.

#### Defining and opening a file:

If we want to store data in a file in secondary memory, we must specify about the file information to the operating system.

- File name (string of char"s, it may contain two parts)
- Data structure (structure is defined as FILE in the library of I/O )
- Purpose (what we want to do with the file we may read /write data)

#### Valid file names

→ input.data  
→ Student.c  
→ Text.out

#### All files should be declared as type FILE before used.

The general format for declaring and Opening

file FILE \*fp;

fp = fopen(“filename”, “mode”);

Example :

FILE \*fp;

fp = fopen(“input.data”. “w”);

Here, fp is a variable of pointer to the data type FILE. Fopen is used to open Input.data file and assigns an identifier to the FILE type pointer fp. This pointer contains all the information

## C Programming & Data Structures

about file and used as a common link between the system and program “w” is mode (write) of opening a file.

- r → opens file for reading only
- w → opens file for writing only
- a → opens file for appending (adding) data.
- r+ → for both reading and writing at the beginning only
- w+ → after writing we may read the file without closing and reopening
- a+ → open a file in read and append mode

- ⇒ When the mode is **writing** new file is created (or) the contents are deleted, if the file is already existed.
- ⇒ When the mode is **appending**, the file is opened and add the contents of new (or) new file also created, if file does not exist.
- ⇒ When the mode is **reading**, if the file existed then the file is opened with the current contents safe otherwise an error occurs.

### Closing a File:

A file must be closed as soon as all operations on it have been completed. This ensures all the information flushed out from the buffers and all links to the file are broken.

Syntax: fclose(file-pointer);

Example: fclose(fp);

### Input/output operations on Files:

getc and putc functions: the simplest file I/o functions are **getc** and **putc**.

- ⇒ „getc” is used to read a char from a file

Example: c=getc(fp1);

- ⇒ „putc” is used to write a char to a file

Example: putc(c1,fp1);

EOF – End of file (when EOF encountered the reading / writing should be terminated)

**Program:** WAP to read data from the keyboard and write it to a file, gain read same data from file, display on the screen.

```
main ( )
{
FILE *fp1;
char c;
clrscr();
fp1 = fopen ("input.data", "w");
while ((c=getchar( ))!=EOF)
putc (c,fp1);
fclose(fp1);
fp1 = fopen ("Input.data",
"r"); while ((c=getc (fp1))!=
EOF) printf("%c",C);
fclose(fp1);
}
```

## C Programming & Data Structures

---

### The getw and putw functions:

These are integer oriented functions. These are similar to above functions and are used to read and write integer values. These are useful when we deal with only integer data. The general format is

```
putw (integer , fp);
getw(fp);
```

### The fprintf and fscanf functions:

Until now, we handle one char/integer at a time. But these functions handle a group of mixed data simultaneously.

Syntax of fprintf is

```
fprintf (fp, "control string", list);
```

Example: fprintf(fp1, "%s %d", name, age);

Syntax of format is,

```
fscanf(fp, "control string", list);
```

Example: fscanf(fp, "%s %d", name, & age);

⇒

**fscanf is used to read list of items from a file**

⇒

**fprintf is used to write a list of items to a file.**

**Program:** WAP to open a file named student and store the data of student name, age, marks. And extended this program read data from this file and display the file on the screen with total marks.

```
#include
<stdio.h> main ( )
{
FILE X fp1; int age, marks, I
total =0;
char name [10];
printf( "enter file name");
scanf ("%s", filename);
fp1 = fopen (filename, "w");
printf( "input student data");
for (i=1; i<=3; i++)
{
fscanf(stdin, "%s %d %d", name, & age, & marks);
fprintf(fp, , "%s %d %d", name, age, marks)"
}

fclose(fp1);
fp=fopen(filename, "r");
printf("student name, age, marks,
total"); for (i=1; i<=3;I++)
{
fscanf(fp, "%s %d %d", name, & age, & marks);
total = total + marks;
fprintf(stdout, "%s %d %d%d", name, age, marks, total);
}
Fclose(fp);
}
```

## C Programming & Data Structures

### Block I/O operations

Some applications involve the use of data files to store blocks of data, where each block consists of a fixed number of contiguous bytes. Each block will generally represent a complex data structure, such as a structure or an array. For example, a data file may consist of multiple structures having the same composition, or it may contain multiple arrays of the same type and size. For such applications it may be desirable to read the entire block from the data file, or write the entire block to the data file, rather than reading or writing the individual components (i.e., structure members or array elements) within each block separately.

The library functions `fread` and `fwrite` are intended to be used in situations of this type. These functions are often referred to as unformatted read and write functions. Similarly, data files of this type are often referred to as unformatted data files. Each of these functions requires four arguments: a pointer to the data block, the size of the data block, the number of data blocks being transferred, and the stream pointer.

Format :

```
int fread(void *pinaarea,int elementsize,int count,FILE* sp); int  
fwrite(void *poutarea,int elementsize,int count,FILE* sp);
```

### Error handling during I/o operations:

- During I/o operations on a file an error may occur. The error may be,
- Trying to read beyond the end-of-file
- Device overflow
- Trying to use a file that has not been opened.
- If the modes are not mentioned properly
- Open a file with invalid name
- Try to write contents to a write protected file

***feof*** and ***ferror*** functions help us to detect I/o errors in the file

⇒ ***feof*** is used to test an end of file condition . It takes a FILE pointer as its only argument and returns a non zero integer value if all the data has been read, and returns zero otherwise.

Example:     if (feof(fp))  
                  printf("end of data");

⇒ ***ferror*** reports the status of the file indicated . It also takes a FILE pointer as its argument and returns a non zero integer if an error has been detected up to that point, during processing , otherwise returns 0.

if (ferror(fp)!=0)  
  printf("an error has ocured");

This prints an error message, if reading is not successful.

⇒ When ever a file is opened using ***fopen***, a file pointer is returned. If the file cannot be opened for some reason, then it returns a NULL pointer. This is used to test whether a file has been opened / not.

Example:     if(fp== NULL);  
                  pf("file could not be opened");

## C Programming & Data Structures

---

### Random Access to files:

Until now we read and write data to a file sequentially. In some situations we need only a particular part of file and not other parts. This can be done with the help of *fseek*, *ftell*, *rewind* functions.

**ftell:** It takes a file pointer and returns number of type long , that corresponds to the current position. This is useful in saving the current position of file.

Example: `n=ftell(fp);`

*n* would give the relative offset (in bytes) of current position. i.e., *n* bytes have all ready been read/ write.

**rewind:** It takes file pointer and resets the position to the start of the file.

Example: `rewind(fp);`

**fseek:** This function is used to move file position to a desired location within file.

Syntax: `fseek (file pointer, offset, position)`

Example: `fseek (Fp1, 0L,);` → go to the beginning  
`Fseek(fp1,m,1)` → go forward by m bytes

**Position** 0 — beginning

1— current position

2— end of file

<b>Offset</b>	Positive	— move forward
	Negative	—move backward
	OL	— no more (stay)

If the *fseek* file operation is successful, it returns *0* else returns *-1* (error occur)

**Program:** WAP to create a new file with the content A, B .....Z. using *ftell* and *fseek* print every fifth character and its position.

```
#include
<stdio.h> main ( )
{
FILE * fp;
long n;
char c;
fp = fopen("randomfile",
"w"); while (cc=getchar ( ) !=
EOF) putc (c,fp);
pf("No. of chars entered = %ld\n", ftell(fp));
fclose(fp);
fp=fopen ("Randomfile",
"r") n=OL;
while (feof(fp)= =0)
{
fseek (fp,n,0);
pf("position of %c is %ld", getc(fp),ftell));
n=n+5L;
}
Puchar(,,\n");
```



## C Programming & Data Structures

---

```

fseek(fp,-1L,2); // position to last char
do
{
    putchar (getc(fp));
} while (!fseek(fp,-
2L,1)); fclose(fp);
}

```

### Command line arguments:

It is a parameter supplied to a program when the program is invoked / executed.

Example: c: > program File-x File-y

Here we copy the contents of file -x to file-y program is the file name where the executable code is stored. This eliminates the need of the program to request the user to enter the filenames during execution.

As we know „main“ function is marked the beginning of the program. This „main“ can take two arguments **argc** and **argv** and information contained in the command line is passed on to the program through these arguments.

→ Argc – is an argument counter (counts the No. of arguments on the command line)

→ Argv – is an argument vector (array of char, pointers which points command line)

The size of this array will be equal to the value of argc. In the above example argc is 3, argv are as follows.

```

argv[0] → Program
argv[1] → File -x
argv[2] → File -y
main (int argc, char * argv[3]) → to access command line arguments
{
    .....
    .....
    .....
}

```

Example:

**Program:** WAP that will receive file name and a line of text as command line arguments and print the text file.

```

#include <stdio.h>
main (int argc, char *argv [])
{
    FILE * fp;
    int i;
    char word[15];
    fp= fopen (argv[1], "w");
    pf("no. of line in command line %ld",
    argc); for (i=2; i<argc; i++)
    fprintf(fp,"%s", argv[i]);
    fclose(fp);
    printf("contents of %s file", argv[1]);
}

```

---



## C Programming & Data Structures

---

```
fp=fopen (argv[1], "r");
for (i=2; i<argc; i++)
{
    Fscanf(fp, "%s", word);
    Printf("%s", word);
}
Fclose(fp);
}
```

Output:

```
c> Prog20 text AAAAAA BBBBBB CCCCCC DDDDDD
No. of arguments in command line = 6
Contents of text file
AAAAA BBBBBB CCCCCC DDDDD
```

### Dynamic Memory Allocation and Dynamic Structures

Dynamic allocation is a pretty unique feature to C (amongst high level languages). It enables us to create data types and structures of any size and length to suit our programs need within the program.

There are two common applications of this:

- dynamic arrays
- dynamic data structure *e.g.* linked lists

### Malloc, Sizeof, and Free

The Function malloc is most commonly used to attempt to ``grab" a continuous portion of memory. It is defined by:

```
void *malloc(size_t number_of_bytes)
```

That is to say it returns a pointer of type void \* that is the start in memory of the reserved portion of size number\_of\_bytes. If memory cannot be allocated a NULL pointer is returned. Since a void \* is returned the C standard states that this pointer can be converted to any type. The size\_t argument type is defined in stdlib.h and is an **unsigned type**.

```
char *cp;          cp = malloc(100);
```

attempts to get 100 bytes and assigns the start address to cp. Also it is usual to use the sizeof() function to specify the number of bytes

```
int *ip;           ip = (int *) malloc(100*sizeof(int));
```

Some C compilers may require to cast the type of conversion. The (int \*) means coercion to an integer pointer. Coercion to the correct pointer type is very important to ensure pointer arithmetic is performed correctly. I personally use it as a means of ensuring that I am totally correct in my coding and use cast all the time..size of can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function.

```
int i;
```

---

## C Programming & Data Structures

---

```

struct COORD
{
    float
    x,y,z };
typedef struct COORD PT;

sizeof(int), sizeof(i),
sizeof(struct COORD) and
sizeof(PT) are all ACCEPTABLE

```

In the above we can use the link between pointers and arrays to treat the reserved memory like an array. **i.e** we can do things like:

```

ip[0] =
100; or
for(i=0;i<100;++i) scanf("%d",ip++);

```

When you have finished using a portion of memory you should always free() it. This allows the memory **freed** to be available again, possibly for further malloc() calls. The function free() takes a pointer as an argument and frees the memory to which the pointer refers.

### Calloc and Realloc

There are two additional memory allocation functions, Calloc() and Realloc(). Their prototypes are given below:

```
void *calloc(size_t num_elements, size_t element_size);
```

```
void *realloc( void *ptr, size_t new_size);
```

malloc does not initialize memory (to **zero**) in any way. If you wish to initialize memory then use calloc. Calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between calloc and malloc in that calloc takes the number of desired elements, num\_elements, and element\_size as two individual arguments. Thus to assign 100 integer elements that are all initially zero you would do

```

int *ip;
ip = (int *) calloc(100, sizeof(int));

```

Realloc is a function which attempts to change the size of a previous allocated block of memory. The new size can be larger or smaller. If the block is made larger then the old contents remain unchanged and memory is added to the end of the block. If the size is made smaller then the remaining contents are unchanged. If the original block size cannot be resized then realloc will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You **must** use this new value. If new memory cannot be reallocated then realloc returns NULL. Thus to change the size of memory allocated to the \*ip pointer above to an array block of 50 integers instead of 100, simply do

```
ip = (int *) realloc( ip, 50);
```

## C Programming & Data Structures

### Unit VII

#### Sorting methods

Sorting is a process in which records are arranged in ascending or descending order. In real life problems across several examples of such sorted information. For example, the telephone directory contains names of the persons and the phone numbers are written according to ascending alphabets. The records of the list of these telephone holders are to be sorted by the name of the holder. By using this directory, we can find the telephone number of any person easily. Sort method has great importance in data structures.

For example, consider the five numbers 5 9 7 4 1

The above numbers can be sorted in ascending or descending order. Ascending order ( 0 to n ) : 1 4 5 7 9

Descending order ( n t 0 ) : 9 7 5 4 1

Sorting is classified in following types

- 1) Bubble sort
- 2) Selection sort
- 3) Quick sort
- 4) Insertion sort
- 5) Merge sort

#### Bubble sort

The bubble sort is an example of exchange sort. In this method, repetitive comparison is performed among elements and essential swapping of elements is done. Bubble sort is commonly used in sorting algorithms. It is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. It is different from the selection sort. Instead of searching the minimum element and then applying swapping, two records are swapped instantly upon noticing that they are not in order.

**Bubble\_Sort ( A [ ] , N )**

Step 1: Repeat for P = 1 to N - 1

    Begin

Step 2: Repeat for J = 1 to N - P

    Begin

Step 3: If ( A [ J ] < A [ J - 1 ] )

        Swap ( A [ J ] , A [ J - 1 ] )

    End For

End For

Step 4: Exit

## C Programming & Data Structures

**Original List**

74	39	35	97	84
----	----	----	----	----

**After Pass 1**

39	35	74	84	97
----	----	----	----	----

**After Pass 2**

35	39	74	84	97
----	----	----	----	----

**After Pass 3**

35	39	74	84	97
----	----	----	----	----

**After Pass 4**

35	39	74	84	97
----	----	----	----	----

### Complexity of Bubble\_Sort

The complexity of sorting algorithm is depends upon the number of comparisons that are made. Total comparisons in Bubble sort is

$$n(n-1)/2 \approx n^2/2 - n/2$$

$$\text{Complexity} = O(n^2)$$

### Selection sort:

The selection sort is nearly the same as exchange sort. Assume that we have a list on  $n$  elements. By applying selection sort, the first element is compared with all remaining  $(n-1)$  elements. The lower element is placed at the first location. Again, the second element is compared with remaining  $(n-1)$  elements. At the time of comparison, the smaller element is swapped with larger element. In this type, entire array is checked for smallest element and then swapping is done.

Selection\_Sort ( A [ ], N )

Step 1 : Repeat For K = 0 to N - 2

    Begin

Step 2 :     Set POS = K

Step 3 :     Repeat for J = K + 1 to N - 1

        Begin

            If A[ J ] < A [ POS ]

## C Programming & Data Structures

Set POS = J

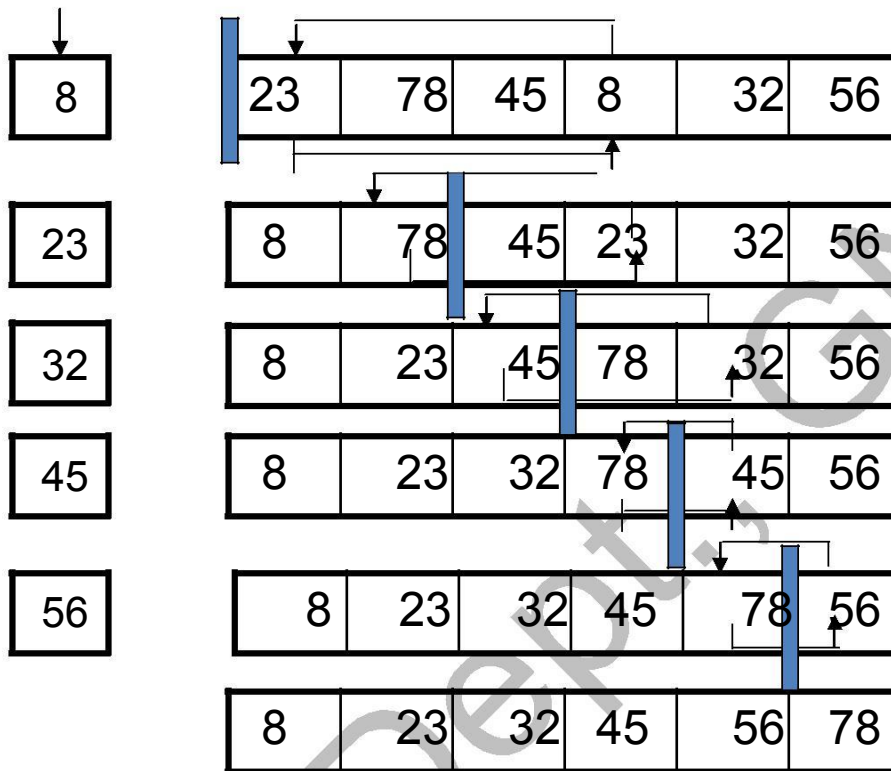
End For

Step 5 : Swap A [ K ] with A [ POS ]

End For

Step 6 : Exit

### Smallest



### Quick sort

It is also known as partition exchange sort. It was invented by CAR Hoare. It is based on partition. The quick sort divides the main list into two sublists. For example, suppose a list of X elements for sorting. The quick sort technique marks an element in the list. The marked element is called pivot or key. Consider the element J. Then it shifts all the elements less than J to left of value J. All the elements greater than J are shifted to the right side. The key element is at the middle of both the lists. It is not compulsory that the selected key element must be at the middle. Any element from the list can act as key element. However, for best performance preference is given to middle elements. Time consumption of the quick sort depends on the location of the key in the list.

## C Programming & Data Structures

### Algorithm for Quick\_Sort

- set the element A [ start\_index ] as pivot.
- rearrange the array so that :
  - all elements which are less than the pivot come left ( before ) to the pivot.
  - all elements which are greater than the pivot come right ( after ) to the pivot.
- recursively apply quick-sort on the sub-list of lesser elements.
- recursively apply quick-sort on the sub-list of greater elements.
- the base case of the recursion is lists of size zero or one, which are always sorted.

### Complexity of Quick Sort

Best Case :  $O(n \log n)$

Average Case :  $O(n \log n)$

Worst Case :  $O(n^2)$

### Original-list of 11 elements

8	3	2	11	5	14	0	2	9	4	20
---	---	---	----	---	----	---	---	---	---	----

Set list [ 0 ] as pivot

pivot

8	3	2	11	5	14	0	2	9	4	20
---	---	---	----	---	----	---	---	---	---	----

Rearrange ( partition ) the elements

into two sub lists : pivot

4	3	2	2	5	0
---	---	---	---	---	---

8
---

11	9	14	20
----	---	----	----

Sub-list of  
lesser elements

Sub-list of  
greater elements

**Apply Quick-sort  
recursively  
on sub-list**

**Apply Quick-sort  
recursively  
on sub-list**

## C Programming & Data Structures

### Insertion sort

In insertion sort the element is inserted at an appropriate place. For example, consider an array of  $n$  elements. In this type also swapping of elements is done without taking any temporary variable. The greater numbers are shifted towards end locations of the array and smaller are shifted at beginning of the array.

Insertion\_Sort ( A [ ], N )

Step 1 : Repeat For  $K = 1$  to  $N - 1$

Begin

Step 2 : Set  $Temp = A [ K ]$

Step 3 : Set  $J = K - 1$

Step 4 : Repeat while  $Temp < A [ J ]$  AND  $J \geq 0$

Begin

Set  $A [ J + 1 ] = A [ J ]$

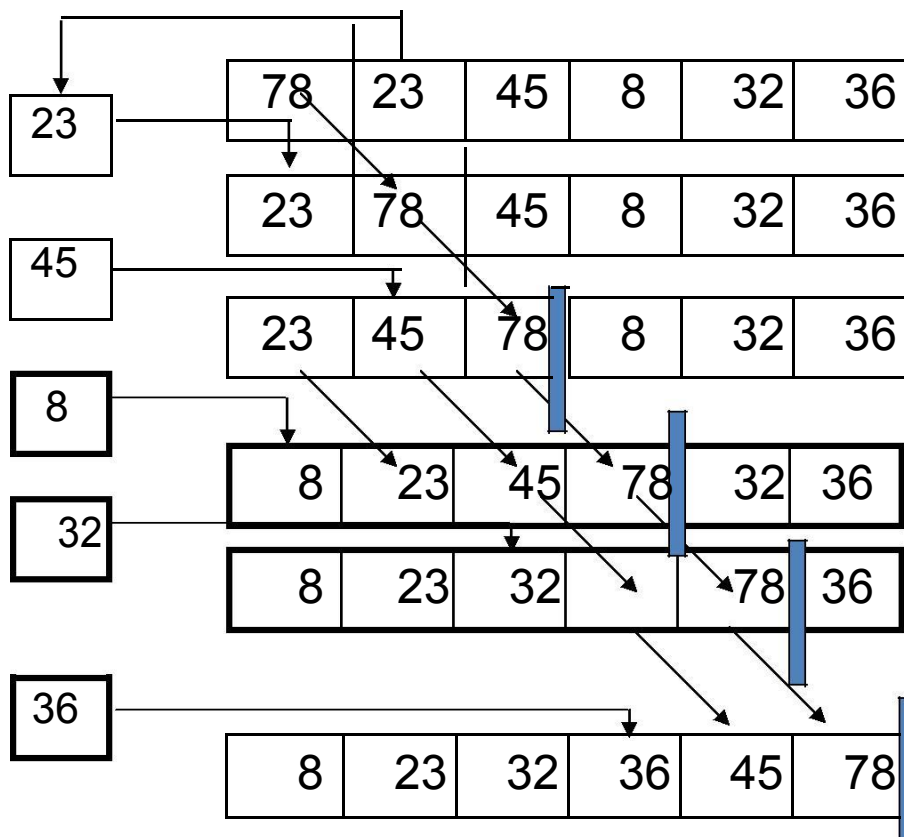
Set  $J = J - 1$

End While

Step 5 : Set  $A [ J + 1 ] = Temp$

End For

Step 4 : Exit



## C Programming & Data Structures

---

### Complexity of Insertion Sort

Best Case:  $O(n)$

Average Case:  $O(n^2)$

Worst Case:  $O(n^2)$

### Merge Sort ( Divide and conquer )

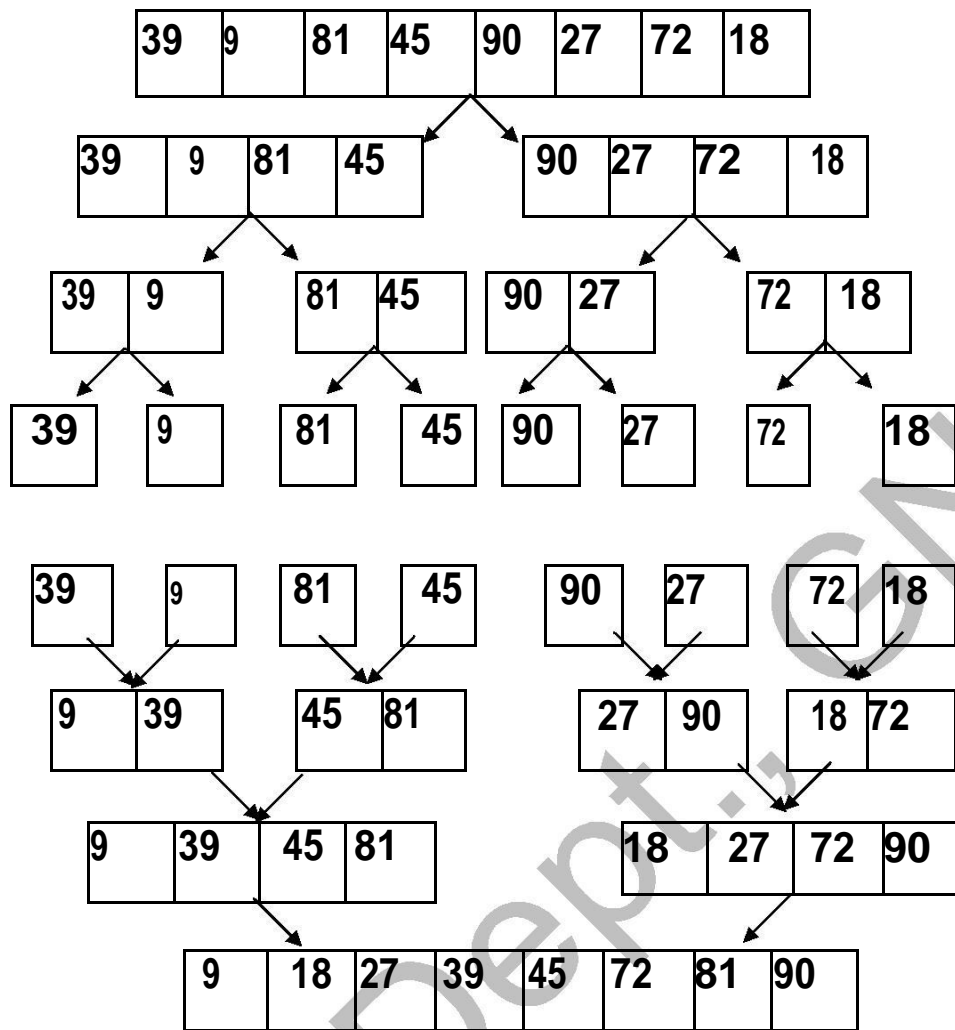
Merge sort technique sorts a given set of values by combining two sorted arrays into one larger sorted array. A small list will take fewer steps to sort than a large list. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. You only have to traverse each list once if they're already sorted.

#### Merge\_sort Algorithm

1. If the list is of length 0 or 1, then it is already sorted.  
Otherwise:+
2. Divide the unsorted list into two sublists of about half the size.
3. Sort each sublist recursively by re-applying merge sort.
4. Merge the two sublists back into one sorted list.



## C Programming & Data Structures



**Merge the elements to sorted array**

Time complexity

Worst case -  $O(n \log n)$

Best case -  $O(n \log n)$  typical,  $O(n)$  natural variant

Average case -  $O(n \log n)$

## C Programming & Data Structures

---

### Searching methods

Searching and sorting are two of the most fundamental and widely encountered problems in computer science. In this handout, we describe four algorithms for search. Given a collection of objects, the goal of search is to find a particular object in this collection or to recognize that the object does not exist in the collection. Often the objects have key values on which one searches and data values which correspond to the information one wishes to retrieve once an object is found. For example, a telephone book is a collection of names (on which one searches) and telephone numbers (which correspond to the data being sought). For the purposes of this handout, we shall consider only searching for key values (e.g., names) with the understanding that in reality, one often wants the data associated with these key values. The collection of objects is often stored in a list or an array. Given a collection of  $n$  objects in an array  $A[1 \dots n]$ , the  $i$ -th element  $A[i]$  corresponds to the key value of the  $i$ -th object in the collection. Often, the objects are sorted by key value (e.g., a phone book), but this need not be the case. Different algorithms for search are required if the data is sorted or not. The input to a search algorithm is an array of objects  $A$ , the number of objects  $n$ , and the key value being sought  $x$ . In what follows, we describe four algorithms for search.

#### Linear Search

Suppose that the given array was not necessarily sorted. This might correspond, for example, to a collection exams which have not yet been sorted alphabetically. If a student wanted to obtain her exam score, how could she do so? She would have to search through the entire collection of exams, one-by-one, until her exam was found. This corresponds to the unordered linear search algorithm.

```
Linear-Search[A, n,
x] 1 for i 1 to n
2 do if A[i] = x
3 then return i
4 else i = i + 1
5 return "x not found"
```

Note that in order to determine that an object does not exist in the collection, one needs to search through the entire collection. Now consider the following array:

```
i  1  2  3  4  5  6  7  8
A 34 16 12 11 54 10 65 37
```

Consider executing the pseudocode Linear-Search[A, 8, 54]. The variable  $i$  would initially be set to 1, and since  $A[1]$  (i.e., 34) is not equal to  $x$  (i.e., 54),  $i$  would be incremented by 1 in Line 4. Since  $A[2] \neq x$ ,  $i$  would again be incremented, and this would continue until  $i = 5$ . At this point,  $A[5] = 54 = x$ , so the loop would terminate and 5 would be returned in Line 3.

Now consider executing the pseudocode Linear-Search[A, 8, 53]. Since 53 does not exist in the array,  $i$  would continue to be incremented until it exceeded the bounds of the for loop, at which point the for loop would terminate. In this case, "x not found" would be returned in Line 5.

## C Programming & Data Structures

---

### Binary Search

Now consider the following idea for a search algorithm using our phone book example. Select a page roughly in the middle of the phone book. If the name being sought is on this page, you're done.

If the name being sought is occurs alphabetically before this page, repeat the process on the "first half" of the phone book; otherwise, repeat the process on the "second half" of the phone book. Note that in each iteration, the size of the remaining portion of the phone book to be searched is divided in half; the algorithm applying such a strategy is referred to as binary search. While this may not seem like the most "natural" algorithm for searching a phone book (or any ordered list), it is provably the fastest. This is true of many algorithms in computer science: the most natural algorithm is not necessarily the best!

```

Binary-Search[A, n, x]
1 low  1
2 high n
3 while low ≤ high
4 do mid ← b(low + high)/2
5 if A[mid] = x
6 then return mid
7 elseif A[mid] < x
8 then low ← mid + 1
9 else high ← mid - 1
10 return "x not found"

```

Again consider the following array:

```

i  1  2  3  4  5  6  7  8
A 10 11 12 16 34 37 54 65

```

Consider executing the pseudocode Binary-Search[A, 8, 34]. The variable high is initially set to 8, and since low (i.e., 1) is less than or equal to high, mid is set to  $b(\text{low} + \text{high})/2 = b(9)/2 = 4$ . Since  $A[\text{mid}]$  (i.e., 16) is not x (i.e., 34), and since  $A[\text{mid}] < x$ , low is reset to  $\text{mid} + 1$  (i.e., 5) in Line 8. Since  $\text{low} \leq \text{high}$ , a new  $\text{mid} = b(5 + 8)/2 = 6$  is calculated, and since  $A[\text{mid}]$  (i.e., 37) is greater than x, high is now reset to  $\text{mid} - 1$  (i.e., 5) in Line 9. It is now the case that low and high are both 5; mid will then be set to 5 as well, and since  $A[\text{mid}]$  (i.e., 34) is equal to x,  $\text{mid} = 5$  will be returned in Line 6. Now consider executing the pseudocode Binary-Search[A, 8, 33]. Binary-Search will behave exactly as described above until the point when  $\text{mid} = \text{low} = \text{high}$ . Since  $A[\text{mid}]$  (i.e., 34) is greater than x (i.e., 33), high will be reset to  $\text{mid} - 1 = 4$  in Line 9. Since it is no longer the case that  $\text{low} \leq \text{high}$ , the while loop terminates, returning "x not found" in Line 10.

```

bool binSearch(int array[], int key, int left, int right) {
    mid = left + (right-left)/2;
    if (key < array[mid])
        return binSearch(array, key, left, mid-1);
    else if (key > array[mid])
        return binSearch(array, key, mid+1, right);
    else if (key == array[mid])
        return TRUE; // Found
    return FALSE; // Not Found
}

```

# C Programming & Data Structures

## Unit VIII

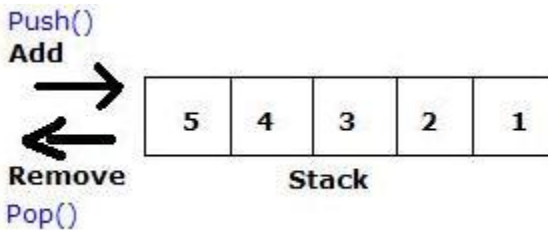
### Stacks

A stack is a homogeneous collection of items of any one type, arranged linearly with access at one end only called top.

This means the data can be added or removed only from top.

Formally this type of stack is called as Last In First Out(LIFO).

Data is added to the stack using push operation, and removed using pop operation.



#### Stack implemented as an array:

One of the 2 ways to implement a stack is by using one dimensional array. When implemented in this way data is stored in the array, and Top is an Integer value, which contains the array index for the top of the stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of the current top of the stack.

An empty stack is indicated by setting top equal to -1.

#### Operations on Stack

Suppose maximum size of stack is „n“

Operation	Description	Requirement
<b>Push</b>	Adds or pushes an item on to the stack	No. of items on the stack should be less than „n“.
<b>Pop</b>	Removes an item from the Stack	No of items in the stack must be greater than 0
<b>Top</b>	Returns the value of the item at the top of the stack.	Note: Item is not removed
<b>IsEmpty</b>	It returns true if the stack is empty and false if it is not	
<b>IsFull</b>	It returns true if the stack is full and false if it is not	

Suppose we have a stack that can hold letters, named as `stack`. We begin with `stack` empty:

```
-----
Stack
```

## C Programming & Data Structures

Now, let's perform `Push(stack, A)`, giving:

```
-----
| A | <-- top
-----
stack
```

Again, another push operation, `Push(stack, B)`, giving:

```
-----
| B | <-- top
-----
| A |
-----
stack
```

Now let's remove an item, `letter = Pop(stack)`, giving:

```
-----
| A | <-- top      | B |
-----
stack              letter
```

And finally, one more addition, `Push(stack, C)`, giving:

```
-----
| C | <-- top
-----
| A |
-----
stack
```

Notice that the stack enforces a certain *order* to the use of its contents, i.e., the *Last* thing *In* is the *First* thing *Out*. Thus, we say that a stack enforces **LIFO** order.

### • Implementing a stack with an array:

Implementing this stack in the C programming language.

First, if we want to store letters, we can use type `char`. Next, since a stack usually holds a bunch of items with the same type (e.g., `char`), we can use an array to hold the contents of the stack.

Consider this array of characters, named `contents`, to hold the contents of the stack. At some point we'll have to decide how big this array is, normal array has a fixed size.

Let's choose the array to be of size 4 for now. So, an array getting **A**, then **B**, will look like:

```
-----
| A | B |   |   |
-----
  0 1 2 3
contents
```

## C Programming & Data Structures

We need to keep track of the *top* of the stack since not all of the array holds stack elements. *top* will hold the array index of the element at the top of the stack.

### Example:

Again suppose the stack has (A,B) in it already...

stack (made up of 'contents' and 'top')

```

-----
| A | B |   |   |   | 1 |
-----
0   1   2   3   top
contents

```

Since **B** is at the top of the stack, the value *top* stores the index of **B** in the array (i.e., 1).

Now, suppose we push something on the stack, `Push(stack, 'C')`, giving:

stack (made up of 'contents' and 'top')

```

-----
| A | B | C |   |   | 2 |
-----
0   1   2   3   top
contents

```

(Note that both the *contents* and *top* part have to change.)

So, a sequence of pops produce the following effects:

```

letter = Pop(stack)
stack (made up of 'contents' and 'top')
-----
| A | B |   |   |   | 1 | | C |
-----
0   1   2   3   top   letter
contents

```

```

letter = Pop(stack)
stack (made up of 'contents' and 'top')
-----
| A |   |   |   |   | 0 | | B |
-----
0   1   2   3   top   letter
contents

```

```

letter = Pop(stack)
stack (made up of 'contents' and 'top')
-----
|   |   |   |   |   | -1 | | A |
-----
0   1   2   3   top   letter
contents

```

We can see what value *top* should have -1 when it is empty.

If we apply the following set of operations

## C Programming & Data Structures

---

1. Push(stack, 'D')
2. Push(stack, 'E')
3. Push(stack, 'F')
4. Push(stack, 'G')

### Result:

stack (made up of 'contents' and 'top')

-----	-----
D   E   F   G	3
-----	-----
0    1    2    3	top
contents	

### Implement Stack using Arrays.

#### 1. Push operation:

Let stack is an array for implementing the Stack data structure. Maxsize specifies the size of the Stack. This algorithm is used for inserting an element in to stack.

- a. [Check for stack overflow]

If Top=Maxsize-1, then display message “Stack Overflow” and

- Exit. b. [Increase Top by 1]

Top=Top+1

- c. [Inserts item in stack]

stack[Top]=Item

- d. Exit

#### 2. Pop operation:

This algorithm is used for deleting an item from top of stack.

- a. [Check for stack underflow]

If Top < 0 then Display message “Stack Underflow “ and exit.

- b. [Decrement the Top pointer by

1] Top=Top-1

- c. Exit

## Stack Applications

➤ Postponement : Evaluating arithmetic expressions

➤ Prefix: + a b

➤ Infix : a + b

➤ Postfix : a b +

In high level languages, infix cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert a infix notation into postfix notation, then evaluating it.

### Infix to Postfix Conversion

- Rules:
  - Operands immediately go directly to output



## C Programming & Data Structures

---

- Operators are pushed into the stack (including parenthesis)
  - Check to see if stack top operator is less than current operator
  - If the top operator is less than, push the current operator onto stack
  - If the top operator is greater than the current, pop top operator and push onto stack, push current operator onto stack
  - Priority 2: \* /
  - Priority 1: + -
  - Priority 0: (

If we encounter a right parenthesis, pop from stack until we get matching left parenthesis. Do not output parenthesis.

### Conversion of Infix expression to Postfix notation

Suppose Q is an arithmetic expression in Infix notation. This algorithm finds the equivalent postfix expression P. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until Stack is empty.

1. If an operand is encountered, add it to P.
2. If a left parenthesis is encountered, push it onto the Stack.
3. If an operator is encountered, then repeatedly pop from Stack and add P each operator (on the top of Stack) which has the same precedence as or higher precedence than current symbol.
  - i. Add the current symbol on to the stack.
4. If a right parenthesis is encountered, then
  - i. Repeatedly pop from Stack and add to P each operator (on the top of Stack. until a left parenthesis is encountered)
  - ii. Remove the left parenthesis.
5. Exit

### Evaluating postfix expression

This algorithm evaluates postfix expression P using Stack.

1. Scan P from left to right and repeat step 3 and 4 for each element of P until the end of the string is encountered.
2. If an operand is encountered, put it on Stack.
3. If an operator (\*) is encountered, then
  - i. Remove the two top elements of Stack, where A is the top element and B is the next to the top element.
  - ii. Evaluate B (\*) A.
  - iii. Place the result of (ii) back to Stack.
4. Set VALUE equal to the top element on Stack.
5. Exit



## C Programming & Data Structures

### Infix to Postfix Example

$A + B * C - D / E$		
<u>Infix</u>	<u>Stack(bot-&gt;top)</u>	<u>Postfix</u>
a) $A + B * C - D / E$		
b) $+ B * C - D / E$		A
c) $B * C - D / E$		A
d) $* C - D / E$	+	A B
e) $C - D / E$	+	A B
f) $- D / E$	+ *	A B C
g) $D / E$	+ *	A B C *
h) $/ E$	+ -	A B C * D
i) $E$	+ - /	A B C * D
j)	+ - /	A B C * D E
k)		A B C * D E / - +

### Infix to Postfix Example #2

$A * B - (C + D) + E$		
<u>Infix</u>	<u>Stack(bot-&gt;top)</u>	<u>Postfix</u>
a) $A * B - (C + D) + E$	Empty	empty
b) $* B - (C + D) + E$	Empty	A
c) $B - (C + D) + E$	*	A
d) $- (C + D) + E$	*	A B
e) $- (C + D) + E$	Empty	A B *
f) $(C + D) + E$	-	A B *
g) $C + D) + E$	-(	A B *
h) $+ D) + E$	-(	A B * C
i) $D) + E$	-( +	A B * C
j) $) + E$	-( +	A B * C D
k) $+ E$	-	A B * C D +
l) $+ E$	Empty	A B * C D + -
m) $E$	+	A B * C D + -
n)	+	A B * C D + - E
o)	Empty	A B * C D + - E +

### Postfix Evaluation

Operand: push

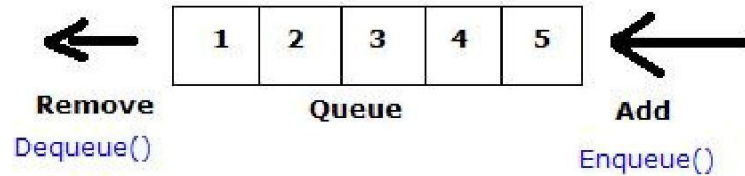
Operator: pop 2 operands, do the math, pop result  
back onto stack

1 2 3 + \*

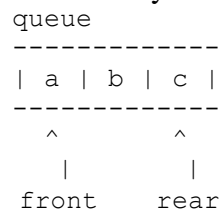
## C Programming & Data Structures

<u>Postfix</u>	<u>Stack( bot -&gt; top )</u>
a) 1 2 3 + *	
b) 2 3 + *	1
c) 3 + *	1 2
d) + *	1 2 3
e) *	1 5 // 5 from 2 + 3
f)	5 // 5 from 1 * 5

### QUEUES



Like a stack, a queue usually holds things of the same type. We usually draw queues horizontally. Here's a queue of characters with 3 elements:



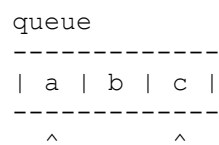
The main property of a queue is that objects go on the *rear* and come off of the *front* of the queue.

Here are the minimal set of operations we'd need for an abstract queue:

- Insert  
Places an object at the *rear* of the queue.
- Delete (or Remove)  
Removes an object from the *front* of the queue and produces that object.
- IsEmpty  
Reports whether the queue is empty or not.

### **Order produced by a queue:**

Queues are useful because they produce a certain order in which the contents of the queue are used. A particular sequence of `Enter` and `Deletes` will do to this queue as follows:



## C Programming & Data Structures

```

      |      |
front  rear

```

**Now, Insert (queue, 'd')...**

```

queue
-----
| a | b | c | d |
-----
      ^           ^
      |           |
front         rear

```

**Now, ch = Delete(queue)...**

```

queue          ch
-----
| b | c | d |  | a |
-----
      ^           ^
      |           |
front         rear

```

The queue enforces a certain *order* to the use of its contents. **First Thing In is the First thing Out (FIFO)**

### Implementing a queue with an array:

Since a queue usually holds a bunch of items with the same type, we could implement a queue with an array. Let's simplify our array implementation of a queue by using an array of a fixed size, MAX\_QUEUE\_SIZE.

One of the things we need to keep track of is the number of elements in the queue, i.e., not all the elements in the array may be holding elements of the queue at any given time.

So far, the pieces of data we need for our array implementation of the queue are:

```

an array
a count

```

We'll start with a queue with 3 elements:

queue (made up of 'contents' and 'count')

```

-----
| a | b | c |   | | 3 |
-----
0   1   2   3   count
contents

```

where **a** is at the *front* and **c** is at the *rear*. Now, we *enter* a new element with: Enter (queue, 'd')...

queue (made up of 'contents' and 'count')

```

-----
| a | b | c | d |   | 4 |
-----

```

## C Programming & Data Structures

---

```
0 1 2 3 count contents
```

**If we remove an element with: `ch = Delete(queue)`**

queue (made up of 'contents' and 'count')

```

-----
|   | b | c | d |   | 3 |   | a |
-----
0     1   2   3   count  ch
Contents

```

Here we cannot insert a new element even though space is available, because the *front* of the queue is no longer at array position 0. One solution would be to move all the elements down one, giving:

queue (made up of 'contents' and 'count')

```

-----
| b | c | d |   |   | 3 |
-----
0     1   2   3   count
contents

```

We **reject** that solution though because it is *too expensive* to move everything down every time we remove an element.

**The Second method is** We can use the index of the element at the front, giving:

queue (made up of 'contents', 'front' and 'count')

```

-----
|   | b | c | d |   | 1 |   | 3 |
-----
0     1   2   3   front  count
contents

```

If we enter another element with: `Insert(queue, 'e')` Currently, the rear of the queue holds 'd' and is at the end of the array.

An alternative would be to use the array in a *circular fashion*. In other words, when we come to the end of the array, we wrap around and use the beginning. Now, with this choice for entering 'e', the fields look like:

queue (made up of 'contents', 'front' and 'count')

```

-----
| e | b | c | d |   | 1 |   | 4 |
-----
0     1   2   3   front  count
contents

```

**After: `ch = Delete(queue)`?**

queue (made up of 'contents', 'front' and 'count')

```

-----
| e |   | c | d |   | 2 |   | 3 |   | b |
-----
0     1   2   3   front  count  Ch

```

## C Programming & Data Structures

---

### contents

Implement Linear Queue using Arrays.

#### **Adding an item to Queue:**

Let queue is an array for implementing the Queue data structure. Max size specifies the size of the Queue. This algorithm is used for inserting an element in to Queue.

1. [Check for Queue overflow]  
If  $\text{Rear} = \text{Maxsize} - 1$  then display "Queue Overflow" and exit.
2. [Increase Rear by 1]  
 $\text{Rear} = \text{Rear} + 1$
3. [Insert item into Queue]  
 $\text{Queue}[\text{Rear}] = \text{Item}$
4. Exit

#### **Deleting an item from Queue:**

1. [Check for Queue underflow]  
If  $\text{Front} > \text{Rear}$  then display "Queue is empty" and exit.
2. [Remove item from Queue]  
 $\text{item} = \text{Queue}[\text{Front}]$
3. [Increment Front by 1]  
 $\text{Front} = \text{Front} + 1$
4. Exit

### **Linked Lists**

A **linked list** is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a datum and a reference (in other words, a *link*) to the next node in the sequence; more complex variants add additional links. This structure allows for efficient insertion or removal of elements from any position in the sequence.

The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

Sequential representation is achieved by using linked representations. Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. Another way of saying this is that in a sequential representation the order of elements is the same as in the ordered list, while in a linked representation these two sequences need not be the same. To access elements in the list in the correct order, with each element we store the address or location of the next element in that list. Thus associated with each data item in a linked representation is a pointer to the next item.

## C Programming & Data Structures

---

This pointer is often referred to as a link. In general a node is a collection of data, data(1), ..., data(n) and links link(1), ..., link(m). Each item in a node is called a field. A field contains either a data item or a link.

It is customary to draw linked lists as an ordered sequence of nodes with links being represented by arrows. Notice that we do not explicitly put in the values of the pointers but simply draw arrows to indicate they are there. This is so that we reinforce in our own mind the facts that (i) the nodes do not actually reside in sequential locations, and that (ii) the locations of nodes may change on different runs. Therefore, when we write a program which works with lists, we almost never look for a specific address except when we test for zero. It is much easier to make an arbitrary insertion or deletion using a linked list rather than a sequential list.

### Dynamic Memory Allocation and Dynamic Structures

Dynamic allocation is a pretty unique feature to C (amongst high level languages). It enables us to create data types and structures of any size and length to suit our programs need within the program.

We will look at two common applications of this:

- dynamic arrays
- dynamic data structure *e.g.* linked lists

### Malloc, Sizeof, and Free

The Function malloc is most commonly used to attempt to ``grab" a continuous portion of memory. It is defined by:

```
void *malloc(size_t number_of_bytes)
```

That is to say it returns a pointer of type void \* that is the start in memory of the reserved portion of size number\_of\_bytes. If memory cannot be allocated a NULL pointer is returned. Since a void \* is returned the C standard states that this pointer can be converted to any type. The size\_t argument type is defined in stdlib.h and is an **unsigned type**.

So:                    char \*cp;                    cp = malloc(100);

attempts to get 100 bytes and assigns the start address to cp. Also it is usual to use the sizeof() function to specify the number of bytes:

```
int *ip;  
ip = (int *) malloc(100*sizeof(int));
```

Some C compilers may require to cast the type of conversion. The (int \*) means coercion to an integer pointer. Coercion to the correct pointer type is very important to ensure pointer arithmetic is performed correctly. I personally use it as a means of ensuring that I am totally correct in my coding and use cast all the time. It is good practice to use sizeof() even if you know the actual size you want -- it makes for device independent (portable) code. sizeof can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function.

SO:

## C Programming & Data Structures

---

```
int i;

struct COORD {float x,y,z};
typedef struct COORD PT;

sizeof(int), sizeof(i),
sizeof(struct COORD) and
sizeof(PT) are all ACCEPTABLE
```

In the above we can use the link between pointers and arrays to treat the reserved memory like an array. *i.e* we can do things like:

```
ip[0] =
100; or
for(i=0;i<100;++i) scanf("%d",ip++);
```

When you have finished using a portion of memory you should always free() it. This allows the memory **freed** to be available again, possibly for further malloc() calls

The function free() takes a pointer as an argument and frees the memory to which the pointer refers.

### Calloc and Realloc

There are two additional memory allocation functions, Calloc() and Realloc(). Their prototypes are given below:

```
void *calloc(size_t num_elements, size_t element_size);
```

```
void *realloc( void *ptr, size_t new_size);
```

Malloc does not initialise memory (to **zero**) in any way. If you wish to initialise memory then use calloc. Calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between calloc and malloc in that calloc takes the number of desired elements, num\_elements, and element\_size, element\_size, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would

```
do: int *ip;
      ip = (int *) calloc(100, sizeof(int));
```

Realloc is a function which attempts to change the size of a previous allocated block of memory. The new size can be larger or smaller. If the block is made larger then the old contents remain unchanged and memory is added to the end of the block. If the size is made smaller then the remaining contents are unchanged.

If the original block size cannot be resized then realloc will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You **must** use this new value. If new memory cannot be reallocated then realloc returns NULL.

Thus to change the size of memory allocated to the \*ip pointer above to an array block of 50 integers instead of 100, simply do:

---

---

## C Programming & Data Structures

---

```
ip = (int *) calloc( ip, 50);
```

### Linked Lists

Let us now return to our linked list example:

```
typedef struct { int value;  
                ELEMENT *next;  
            } ELEMENT;
```

We can now try to grow the list dynamically:

```
link = (ELEMENT *) malloc(sizeof(ELEMENT));
```

This will allocate memory for a new link.

If we want to deassign memory from a pointer use the free()  
function: free(link)

### ALGORITHM:

#### Single Linked List

Inserting a new node at the beginning of the linked list:

1. Allocate memory for the new node.
2. Assign the value to the data field of the new node.
3. Make the link field of the new node points to the starting node of the linked list.
4. Then, change the head pointer to point to the new node.

#### Inserting a new node at the end of the linked list:

1. Allocate memory for the new node.
2. Assign the value to the data field of the new node.
3. If the list is empty, then set the current node as the start node or head of the linked list.
4. If the linked list is not empty, then go to the last node and then insert the new node after the last node.
5. Make the link field of the new node to NULL.

#### Inserting a new node at the specified position:

This algorithm inserts a new node X between two existing nodes A and B.

1. Allocate memory for the new node.
2. Assign the value to the data field of the new node.
3. Set the link field of the new node to point to node B.
4. Set the link field of the node A point to the new node X.

#### Deleting the first node:

1. If the list is not empty, move the head pointer to point to the second node.
  2. Free the first node.
-



## C Programming & Data Structures

---

### Deleting the last node:

1. If the list is not empty, go to the last node.
2. Set the link field of last but one node to NULL.
3. Free the last node.

### Traversal:

1. Read the value to be search.
2. Set the temp pointer to point to the first node.
3. Repeat steps 4 thru 5 until temp points NULL
4. display the data field value of the node
5. change the pointer to next node
6. stop

### Implement Stack using pointers

The working principle of stack is LIFO. Using a linked list is one way to implement a stack so that it can handle essentially any number of elements. The operations performed on a stack are push ( ) and pop ( ). When implementing a stack using linked lists, pushing elements on to the stack and popping elements from the stack is performed at the same end i.e. TOP.

#### Pushing elements to the stack:

1. Allocate memory for the new node.
2. Assign the value to the data field of the new node.
3. Set the link field of the new node to NULL.
4. If the stack is empty, then set TOP pointer points to the new node and exit.
5. Set the link field of the node pointed by TOP to new node.
6. Adjust the TOP pointer points to the new node and exit.

#### Popping elements from the stack:

1. If stack is empty, then display message “stack is empty – no deletion possible” and exit.
2. Otherwise, move the TOP pointer points to the last but one node.
3. Free the last node.

### Implement Queue using pointers

The working principle of queue is FIFO. Using a linked list is one way to implement a queue so that it can handle essentially any number of elements. The operations performed on a queue are insert( ) and delete( ). When implementing a queue using linked lists, inserting elements to the queue is done at one end and deletion of elements from the queue is done at the other end.

#### Inserting a node at the Rear end:

1. Allocate memory for the new node.
-

## C Programming & Data Structures

---

2. Assign the value to the data field of the new node.
3. Set the link field of the new node to NULL.
4. If the queue is empty, then set FRONT and REAR pointer points to the new node and exit.
5. Set the link field of the node pointed by REAR to new node. A
6. Adjust the REAR pointer points to the new node.

### **Deleting a node at the Front end:**

1. If queue is empty, then display message “ Queue is empty – no deletion is possible” and exit.
2. Otherwise, move the FRONT pointer points to the second node.
3. Free the first node.