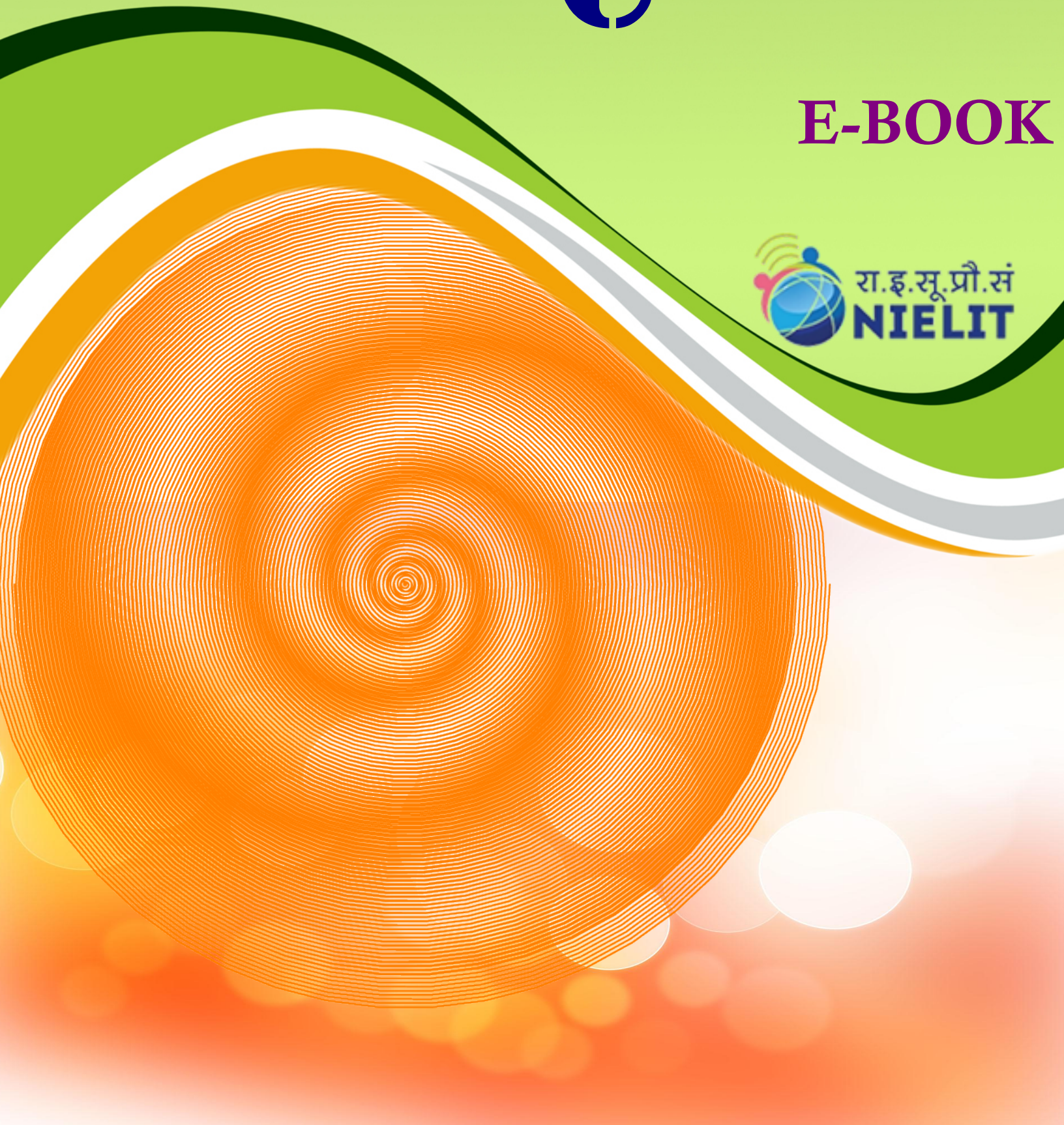
PROBLEM SOLVING THROUGH

C

E-BOOK



UNIT - 1

Introduction to Programming

Introduction

- Problem definition:
  - A problem definition involves the clear identification of the problem in terms of available input parameters and desired solution.
- Approach towards solving the problem:
  - After a problem is identified, the user needs to implement a step-by-step solution in terms of algorithms.
- Graphical representation of problem solving sequence:
  - This step involves representing the steps of algorithm pictorially by using a flowchart.
  - Each component of the flowchart presents a definite process to solve the problem.
- Converting the sequence in a programming language:
  - Converting the graphical sequence of processes into a language that the user and the computer can understand and use for problem solving is called programming.
  - After the program is compiled the user can obtain the desired solution for the problem by executing the machine language version of the program.

Algorithm

- A sequential solution of any program that written in human language, called algorithm.
- Algorithm is first step of the solution process, after the analysis of problem, programmer writes the algorithm of that problem.

Sample Algorithm

- Algorithm to reverse the digits of an integer

Input: Number to be reversed >=N

Output: Reversed number N

Step 1:

- Input the Number be reversed N.

Step 2:

- Assign the value of the number N to M
- Assign the value L = 0
- Store the remainder which obtained by getting the modulus of N and 10 in a variable K

Step 3:

- Divide M by 10
- Assign L equal to Product of L and 10 and add the sum to K

Step 4:

- If M is greater than 0
- Yes: Repeat step 3
- No: Output The reverse of the number N is L

Step 5:

- Stop

- An algorithm is a step-by-step procedure for solving a stated problem using a reasonable amount of time and storage.
- Every problem has some input information and some desired results to be obtained.
- The important aspect is to determine if the information available is really sufficient to solve the problem at hand, and if it is so what are the procedures that should be applied to the given information or data so that the required result can be obtained.
- The development of a proper procedure to solve the problem is called an algorithm.

- The operations that can be included in an algorithm are constrained by the possibility of a computer carrying them out.
- Each operation must be unambiguous about the process to solve the problem.
- The steps in the process must be effective enough to be theoretically done in a finite amount of time and memory.
- An algorithm terminates in a reasonably short time.
- Sample Algorithm:
  - Go to bus stop.
  - Catch bus no. M7-B.
  - Purchase a ticket.
  - Get down at the market bus stop.

Example of Algorithm

- Find out number is odd or even

```
Step 1: start
Step 2: input number
Step 3: rem=number mod 2
Step 4: if rem=0 then
            print "number even"
        else
            print "number odd"
        end if
Step 5: stop
```

What is pseudo code?

- Pseudocode is a simple way of writing programming code in English.
- Pseudocode is not an actual programming language.
- Pseudo code should not include keywords in any specific computer language.
- It should be written as a list of consecutive phrases.
- The user should not use flowcharting symbols but he can draw arrows to show looping processes.

- Indentation can also be used in pseudo code to show the logic.
- The purpose of using pseudo code is that it is easier for humans to understand than conventional programming language code, and that it is a compact and language independent description of the key principles of an algorithm.
- Pseudo code is used to quickly draw the outline of a program.
- It can use any human language.

Example of pseudo code:

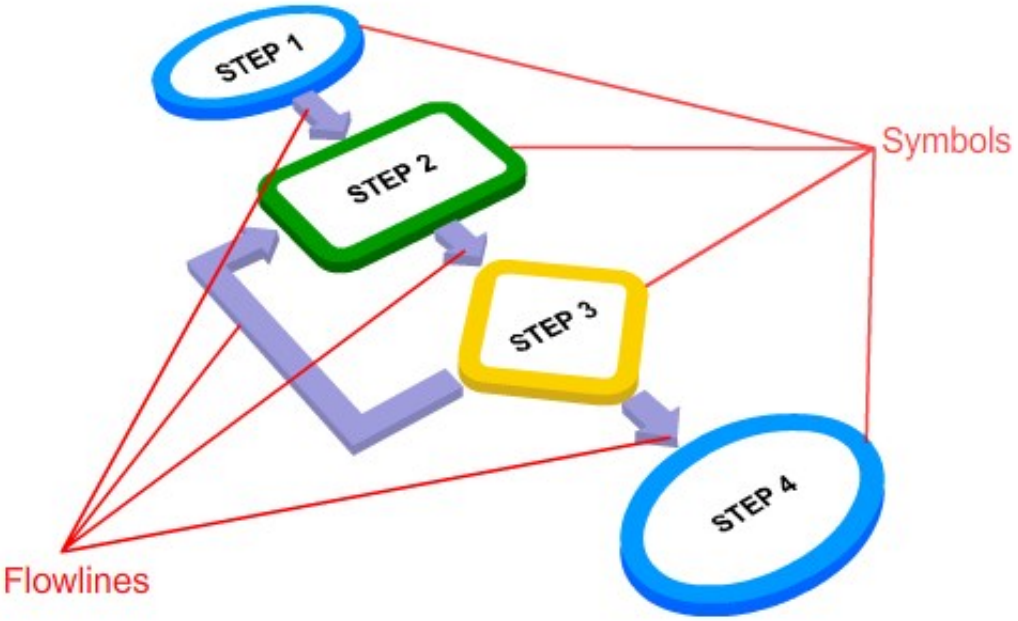
```
Initiate the program

Loop until i = 10

Write "hello world" to the screen

End exit the loop
```

- This is pseudocode.
- It would not work if you ran it through a c compiler, but it explains the structure of the code and how users are planning to implement it.

```
if student's grade is greater than or equal to 60
    print "passed"
else
    print "failed"
end if
```

What is a flowchart?

- A flowchart is a pictorial representation of the steps that are involved in the procedure.
- A flowchart also shows the logical sequence in which the steps are to be performed.
- A flowchart consists of boxes called the symbols and arrows called the flow lines.
- The box depicts the process and the flow line indicates the next step to be performed.



Elements of a flow chart

Terminal Box

- This symbol is used to indicate the beginning or the end of a flowchart.
- When this symbol is used for START, no flow lines can enter it.
- Only one flow line can leave this box.
- When the terminal box is used for STOP, no flow lines can leave this box and any number of flow lines can enter this box.
- There can be only one START symbol in each flowchart.
- However there can be more than one STOP symbol in a flowchart.

Process Box

- The shape of the processing box is a rectangle.
- This symbol represents one or more instructions that perform a processing function in a program.
- Examples of processing functions are addition, subtraction, multiplication, division or moving data to storage or assigning a value.

Input/output Box

- The input/output box is represented by a parallelogram.
- This symbol indicates any function of an input/output device such as keyboard or printer.
- An input/output box makes data available for processing or displays the result of processing on the screen.

Decision Box

- The shape of a decision box is a rhombus.
- This box is used when two quantities need to be compared.
- The decision box is also used for a testing condition.
- Decision box results in two alternative answers to the condition: true or false.
- The equality, less than, less than or equal to, greater than, greater than or equal to, and not equal to operators can be used in the decision box.

Example of Algorithm and flowchart for same problems

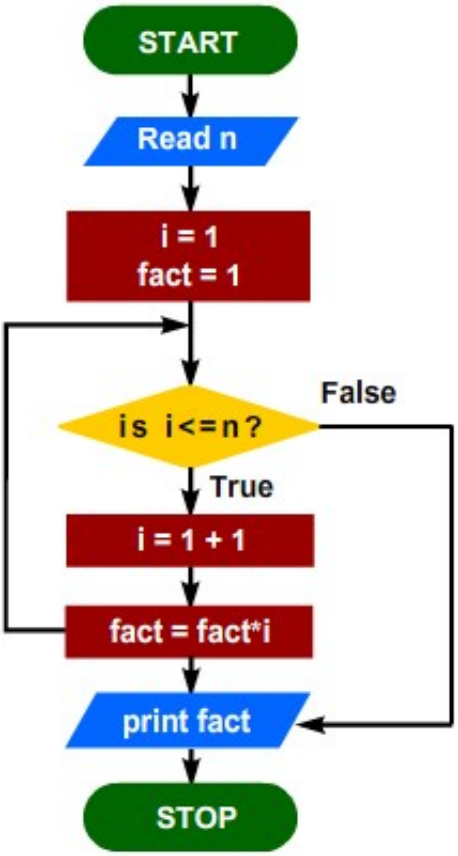
- Algorithm for calculate factorial value of a number:

[Algorithm to calculate the factorial of a number]

Step 1. Start
Step 2. Read the number n
Step 3. [Initialize]
i=1, fact=1
Step 4. Repeat step 4 through 6 until i<=n
Step 5. i=i+1
Step 6. fact=fact*i
Step 7. Print fact
Step 8. Stop

[Process finish of calculate the factorial value of a number]

Flowchart for calculate factorial value of a number



Programming language - Introduction

- A programming language is a set of symbols, grammars and rules with the help of which one is able to translate algorithms to programs that will be executed by the computer.
- The programmer communicates with a machine using programming languages.
- There are many different classifications of programming languages and these programming languages differ in their closeness to the machine and in the way they are structured.
- Most of the programs have a highly structured set of rules.
- The primary classifications of programming languages are:
  - Machine Languages.
  - Assembly Languages.
  - High - level Languages.

Machine Level Language

- Machine language is a collection of binary digits or bits that the computer reads and interprets.
- Machine language is the only language a computer is capable of understanding.
- Machine level language is a language that supports the machine side of the programming or does not provide human side of the programming.
- It consists of (binary) zeros and ones.
- A machine-level language is the lowest form of computer language.
- Each instruction in a program is represented by a numeric code, and numerical addresses are used throughout the program to refer to memory locations in the computer's memory.
- All book keeping aspects of the program are the sole responsibility of the machine-language programmer.
- Finally, all diagnostics and programming aids must be supplied by the programmer.
- Also included as machine-level programs are programs written in microcode (i.e., micro programs).
- Microcode allows for the expression of some of the more powerful machine-level instructions in terms of a set of basic machine instructions.
- Each computer has only one programming language which does not need a translating program – the machine language.
- Machine language programs, the first generation programs, are written at the most basic level of computer operation.

- Because their instruction are directed at this basic level of operation, machine language and assembler language are collectively called low – level language.
- In machine language, instruction are coded as a series of ones and zeroes.
- The machine language programs are cumbersome and difficult to write.
- The machine language is native to that machine and understood directly by the machine.
- The machine language generally has two parts:

Opcode	Operand

- The opcode of machine language tells what function to perform to the computer.
- The operand gives the data on which the operation has to be performed or the location where the data can be found.

Advantages

- Machine – level instructions are directly executable.
- Machine – level language makes most efficient use of computer system resources like storage and register
- Machine language instruction can be used to manipulate individual bits.
- As the machine inherently understands machine instruction, machine languages are very fast.

Disadvantages

- Difficult to program: Programming in machine language is the most difficult kind of programming. Instruction should be encoded as a sequence of incomprehensible 0′s and 1′s which is very difficult.
- Error-prone: In machine language, the programmer has to look into all the activities like memory management, instruction cycle, etc., which diverts his attention from the actual logic of the program. This frequency leads to error.
- Machine dependent: Every computer is different from one another in its architecture. Hence, instructions of one machine will be different from the other.
- As machine – level language are device dependent, the programs are not portable from one computer to another.
- Programming in machine language usually results in poor programmer productivity.
- Programs in machine language are more error prone and difficult to debug.
- Computer storage location must be address directly, not symbolically.
- Machine language requires a high level of programming skill, which increases programmer

training costs.

Assembly Level Language

- A set of instructions for an assembly language is essentially one-to-one with those of machine language.
- Like machine language, assembly language are unique to a computer.
- The big difference is that instead of a cumbersome series of ones and zeroes, assembly languages use easily recognizable symbols called mnemonics, to represent instructions.
- As said before, machine language and assembly language are low-level languages and are dependent on particular machine architecture.
- They are more close to the machine rather than the programmer.
- The only difference between assembly language and machine language is that assembly language is relatively easier to use than machine language.
- In a pure assembly language, each statement produces exactly one machine instruction (one-to-one correspondence between machine instructions and statements in assembly language).
- So an on-line assembly language will produce an n-word machine language program.
- Assembly language which uses mnemonic codes is easier than machine language using binary or hexadecimal codes.
- It is easier to remember ADD, SUB, MUL, or DIV, than their corresponding numerical values in machine language.
- Assembly language uses symbolic names for memory locations while machine language needs numerical values.
- The assembly language can directly test if there is an overflow bit while a higher level language cannot.
- An assembly language can only run on one family of machines (each machine has its own assembly language) while a higher level language can run on many machines.

Advantages

- Assembly language is easier to use than machine language.
- An assembler is useful for detecting programming errors.
- Programmers do not have the absolute address of data items.
- Assembly language encourage modular programming.

Disadvantages

Assembly language programs are not directly executable.

- Assembly language are machine dependent and, therefore, not portable from one machine to another.

- Programming in assembly language requires a higher level of programming skill.

High Level Language

- High level language is a language that supports the human and the application sides of the programming (typical features: ability to logic structuring of the algorithm, cross-platform independence).

- A language is a machine-independent way to specify the sequence of operations necessary to accomplish a task.

- A language can be designed to be express in a concise way, a common sequence of operations.

- A line in a high level language can execute powerful operations, and correspond to tens, or hundreds, of instructions at the machine level.

- Although many programmers prefer assembly language, because it works more efficiently, the power and flexibility of new generation languages have put them beyond low-level languages in terms of both human and computer efficiency.

- Consequently more programming is now done in high-level languages.

- Examples of high-level languages are BASIC, FORTRAN etc.,

- High-level languages are more, like English statements, which are easy to understand and to use.

- High-level languages are more close to the programmer and ease the task of programming.

- Higher-level languages provide a richer set of instructions and support, making the programmer's life even easier.

- Yet before a high-level program can be executed on a given CPU, it must be translated back to machine code.

- A programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that is more or less independent of a particular type of computer.

- Such languages are considered high-level because they are closer to human languages and further from machine languages.

- In contrast, assembly languages are considered low-level because they are very close to machine languages.

Features of high level languages

- High level languages are easily understandable.

- The programs that are developed in high level language are portable.

- In case of high level languages debugging of the code is easy and the program written is not machine dependent.

Advantages of high-level languages include

- Easier to program.
  - The programmer can concentrate on the logic of the program rather than on the register, ports and memory storage.
- Machine-independent.
  - Provided some other machine has the same compiler and libraries, the program can be ported across various platforms.
- Easy maintenance.
  - A program in a high-level language is easier to maintain than assembly language, because it is easier to code and locate bugs in high-level languages.
- Easy to learn.
  - The learning curve for high-level languages is relatively smooth than low-level languages.
- Other than this, high-level languages are more flexible and can be easily documented.

Compilation

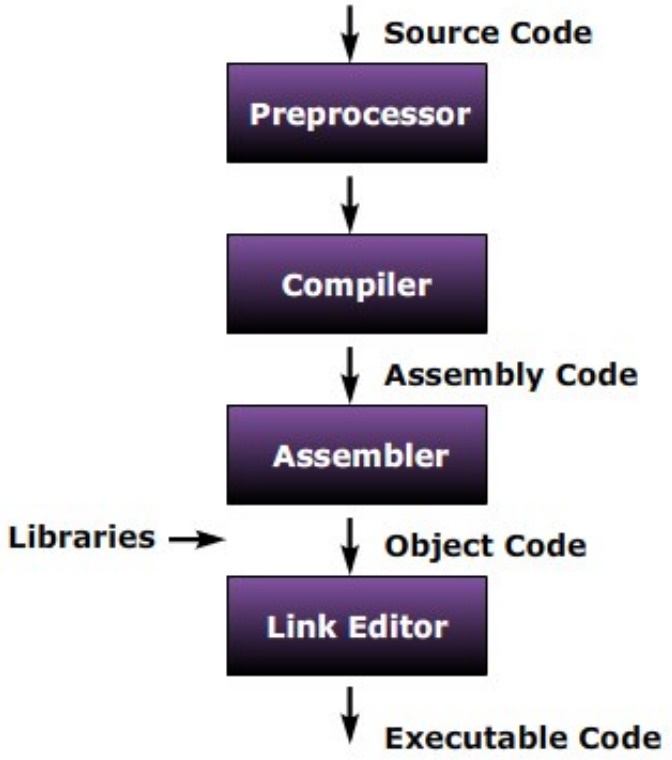
- The compiler program translates the instructions of a high-level language to a machine level language.
- A separate compiler is required for every high-level language.
- High level language is simply a programmer's convenience and cannot be executed in their source.
- The actual high-level program is called a source program.
- It is compiled (translated) to machine level language program called object program for that machine by the compiler.
- Such compilers are called self-resident compilers.
- Compiler compiles the full program and reports the errors at the end.

What is "compilation"?

- The compiler is the tool to convert a program written in a high level language into the sequence of machine instructions required by a specific computer to accomplish the task.
- Users typically control details of the operation of compilers by means of options supplied on the command line, or directives embedded in the program source, but they seldom need to examine the resulting machine language code.
- This process of converting high level language to machine level is called compilation.
- The translation of source code into object code by a compiler.

Compilation Process

- The compilation and execution process of C can be divided in to multiple steps:
  - Preprocessing - Using a Preprocessor program to convert C source code in expanded source code. "#include" and "#define" statements will be processed and replaced actually source codes in this step.
  - Compilation - Using a Compiler program to convert C expanded source to assembly source code.
  - Assembly - Using a Assembler program to convert assembly source code to object code.
  - Linking - Using a Linker program to convert object code to executable code. Multiple units of object codes are linked to together in this step.
  - Loading - Using a Loader program to load the executable code into CPU for execution.


Source Code
Preprocessor
Compiler
Assembly Code
Assembler
Libraries
Object Code
Link Editor
Executable Code

Linking and Loading

What is "linking"?

- After all of the files are compiled, they must be "merged together" to produce a single executable file that the user use to run the program.
- In C, most compiled programs produce results only with the help of some standard programs, known as library files that reside in the computer.
- This process is called linking.
- The result obtained after linking is called the executable file.
- To build an executable file, the linker collects and libraries.
- The linker′s primary function is to bind symbolic names to memory addresses.
- To do this, it first scans the files and concatenates the related file sections to form one large file.
- Then, it makes a second pass on the resulting file to bind symbol names to real memory addresses.
- Linking is the process of taking some smaller executable and joining them together as a single larger executable.
- Loading is loading the executable into memory prior to execution.
- There are two types of linking:
  - Static linking.
  - Dynamic linking.
- Static linking occurs at compilation time; hence it occurs prior to loading a program.
- With static linking the external symbols that are used by the program (e.g. function names) are resolved at compile time.
- Dynamic linking occurs at run time, so it occurs after or at the time of the loading of a program.
- With dynamic linking the symbols are resolved either at loading time, or at run time when the symbol is accessed (lazy binding).

What is "loading”?

- After the files are compiled and linked the executable file is loaded in the computer′s memory for executing by the loader.
- This process is called Loading.
- Program loading is basically copying a program from secondary storage into main memory so it ′s ready to run.

- In some cases, loading us just not copying the data from disk to memory, but also setting protection bits, or arranging for virtual memory map virtual addresses to disk pages.

Testing and Debugging

- Testing involves finding problems in the code. What does one Test for?
- Compilation Errors.
- Syntax errors: The compiler cannot understand user's program because it does not follow the syntax.
- Common syntax errors are:
  - Missing or misplaced; or}
  - Missing return type for a procedure.
  - Missing or duplicate variable declarations.
- Type errors
  - These errors include type mismatch when the user assign a value to a variable and type mismatch between actual and formal parameters.
- Runtime Errors
  - Output errors: These errors result when the program runs but produces an incorrect result.
  - An output error indicates an error in the meaning or logic of the program.
  - Exceptions: These errors occur when the program terminates abnormally.
  - Examples include division by zero and out of memory. Exceptions indicate an error in the semantics or the logic of a program.
- Non-termination errors
  - These errors occur when the program does not terminate as expected, but continues running endlessly.
  - Debugging consists of isolating and fixing the problems.
  - Testing and debugging are necessary stages in the development cycle, and they are best incorporated early in the cycle.
  - Thoroughly testing and debugging individual components makes testing and debugging integrated applications much easier.

What is Documentation?

- Documentation of a program consists of written description of program's specification, its design, coding, operating procedures etc.
- Documentation is either incomplete or inaccurate, making the users and maintenance programmers irritated and annoyed over it.
- Documentation can be broadly classified into two types:
  - Documentation for users.
  - Documentation for maintenance programs.
- The written text and comments that make a program easier for others to understand use and modify.
- The compiler in the compilation process ignores comments.
- It provides information about the steps and procedures.

UNIT - 2

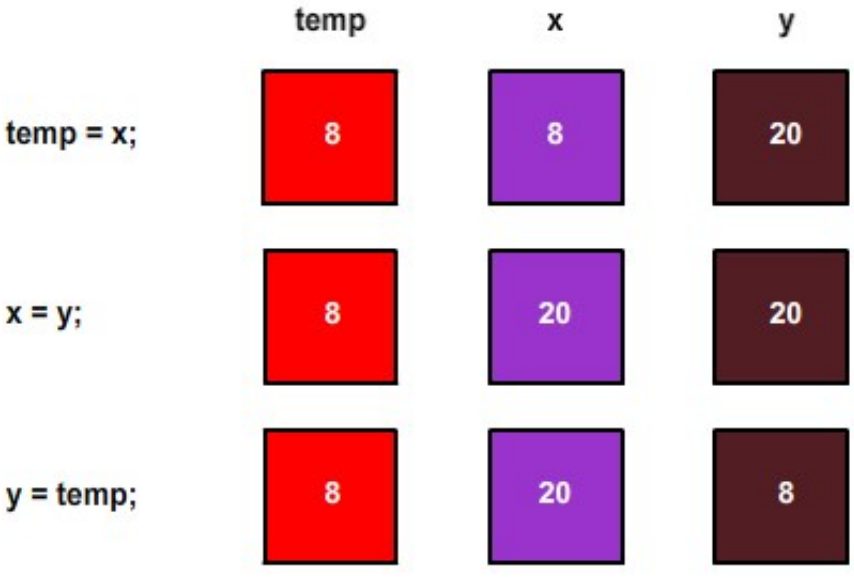
Algorithms for Problem Solving

Exchanging Values of Two Variables

- Problem definition: Exchanging values of two variables.
- Analysis: Two variables x and y contains two different values.
- Swap the values of x and y such that x has y′s value and y has x′s value.
- Solving by example: Let us consider two variables x and y,containing values 8 and 20 respectively.
- The original values of x and y are:

- The requirement is once the algorithm is performed, the results should be

- If you think by just saying,
  - x=y; y=x;
- The value gets swapped, then you are mistaken.
- These instruction are atomic in nature and hence x = y means that the value of ‘x’ is lost.
- So,we have to use a temporary variable,temp to store the value of ‘x’.



- The value of ‘x’ and ‘y’ is swapped.

Algorithm Definition

Step 1: Start.

Step 2: Get the values of x and y.

Step 3: Store x's value to temp. (temp: =x)

Step 4: Store y's value to x. So, x has y's value now (x: =y)

Step 5: Store temp’s value (the value of the old 'x') in y.

Step 6: Stop.

Example Program: To exchange the values of two variables using a temporary variable

```
#include <stdio.h>
 int main()
{
   int x, y, temp;
   printf("Enter the value of x and y\n");
   scanf("%d%d", &x, &y);
   printf("Before Swapping\nx = %d\ny = %d\n",x,y);
   temp = x;
   x    = y;
   y    = temp;
   printf("After Swapping\nx = %d\ny = %d\n",x,y);
   return 0;
}
```

Summation of a Set of Numbers

- Problem Definition: Summation of a set of numbers.
- Problem Analysis: Given a list of ‘n’ numbers, the algorithm should add up the ‘n’ numbers to find the sum of these values.
- Solving by Example: Consider a list of 5 numbers.

- The algorithm takes these numbers as input and should produce the output as 47.

Algorithm Definition

Step 1: Start.

Step 2: Get the values of 'n' numbers that need to be summed up.

Step 3: Initialize sum to zero.

Step 4: Initialize i to one.

Step 5: While i is less than or equal to n, do the following steps:

(a) Read the i^{th} number.

(b) Add the i^{th} number to the current value of sum.

(c) Increment i by 1.

Step 6: Print the value of sum.

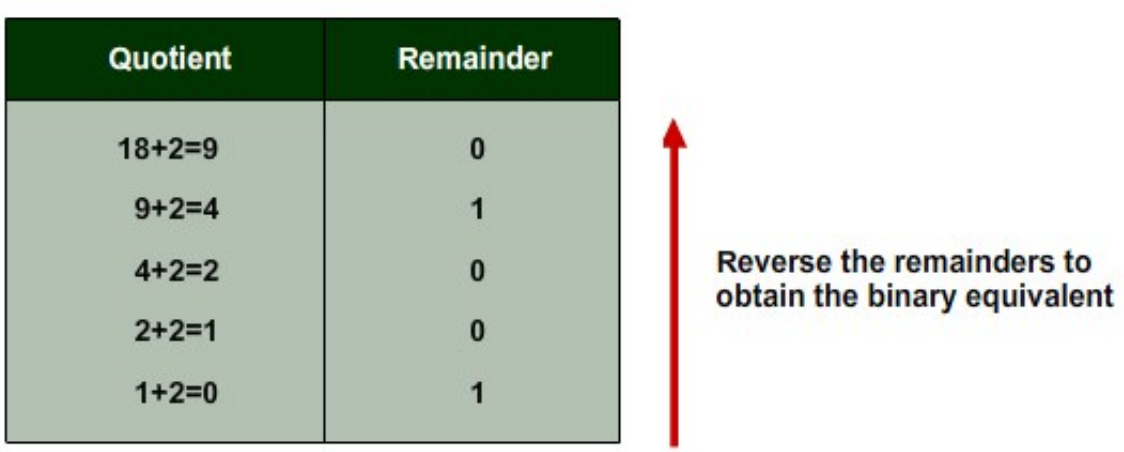
Step 7: Stop.

To Summation of a set of numbers

```
#include <stdio.h>
  int main()
   {
     int n, sum = 0, c, value;
     printf("Enter the number of integers you want to add\n");
     scanf("%d", &n);
     printf("Enter %d integers\n",n);
     for(c = 1; c <= n; c++)
      {
       scanf("%d", &value);
       sum = sum + value;
      }
     printf("Sum of entered integers = %d\n",sum);
     return 0;
   }
```

Decimal Base to Binary Base Conversion

- Problem Definition: Decimal to binary base conversion.
- Problem Analysis: A number with base ‘n’ will contain digits from 0 to n-1.
- Hence decimal base system is 10.
- We use in our normal containing digits from 0-9.
- For example, 922_{10} is a number with base 10.
- A binary number system has a base 2.
- Hence it uses just two numbers i.e. 0 and 1.
- The computer system stores information in the binary number system.
- The conversion of decimal number to binary number is done by dividing the decimal number by 2 repeatedly and by accumulating the remainders obtained, till the quotient becomes zero.
- The reverse of the accumulated remainders is the binary, equivalent the decimal number.
- Solving by Example: Consider the decimal number $(18)_{10}$



Quotient	Remainder
18+2=9	0
9+2=4	1
4+2=2	0
2+2=1	0
1+2=0	1

- Hence $(18)_{10}$ equivalent to $(10010)_2$.

Algorithm Definition

Step 1: Start.

Step 2: Read the decimal number n that has to be converted to binary.

Step 3: Initialize bin to zero.

Step 4: Initialize i to zero.

Step 5: Repeat the following steps until n is greater than zero:

(i) Divide n by 2

(ii) Store the quotient to n

(iii) Add the value (10'* Remainder) to the value i bin:

[This part will take care reversing the accumulated remainders)

(iv) Increment i by 1

Step 6: Print the value of bin.

Step 7: Stop.

To convert a number from decimal base to binary

```
#include <stdio.h>
 int main()
  {
    int n, c, k;
    printf("Enter an integer in decimal number system\n");
    scanf("%d", &n);
    printf("%d in binary number system is:\n", n);
    for (c = 31; c >= 0; c--)
     {
       k = n >> c;
       if (k & 1)
       printf("1");
       else
       printf("0");
     }
   printf("\n");
   return 0;
}
```

Reversing Digits of an Integer

- Problem Definition: Reversing the digit of an integer.
- Problem Analysis: Given a integer number, abcd (say 1234where a=I, b=2andso on), the algorithm should convert the number to dcba.
- Solving by Example: Consider an integer number 7823.
- The reverse of an integer number is obtained by successively dividing the integer by 10 and accumulating the remainders till the quotient becomes less than 10.
- The remainder of a division operation can be found out by the mod function.

	Accumulated	Quotient
7823 Mod 10	3	782
782 Mod 10	32	78
78 Mod 10	328	7
7 Mod 10	3287	0

Algorithm Definition

Step 1: Start.

Step 2: Read the integer n to be reversed.

Step 3: Initialize rev to zero.

Step 4: Write 'n' is greater than 0, does the following step:
(a) Divide the integer n by 10.
(b) Store the quotient to n.
(c) Multiply 10 to the already existing value of rev.
(d) Add the remainder to the value of rev.

Step 5: Print the value of rev.

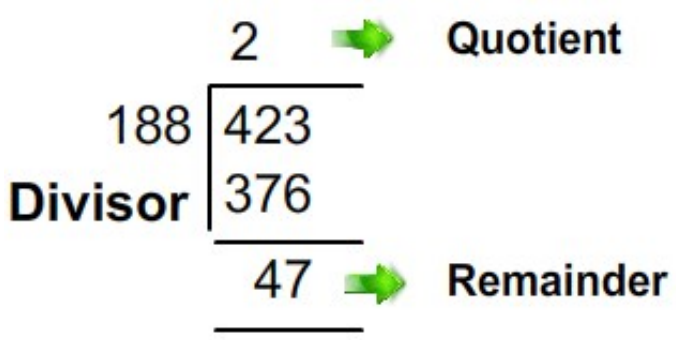
Step 6: Stop.

To reverse the digits of an integer

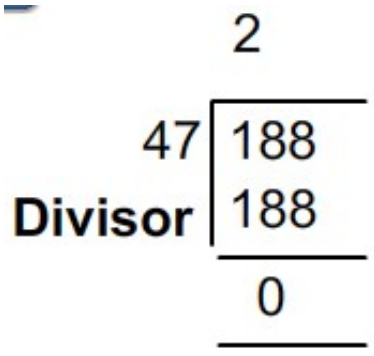
```
#include <stdio.h>
 int main()
{
  int n, reverse = 0;
  printf("Enter a number to reverse\n");
  scanf("%d", &n);
  while(n!= 0)
  {
    reverse = reverse * 10;
    reverse = reverse + n%10;
    n = n/10;
  }
 printf("Reverse of entered number is = %d\n", reverse);
 return 0;
}
```

GCD (Greatest Common Division) of Two Numbers

- Problem Definition: To find the Greatest Common Divisor (GCD)of two numbers.
- Problem Analysis: GCD of two numbers is the greatest common factor of the given two numbers.
- GCD of two numbers is obtained using the following method: Divide the larger of the two numbers by the smaller one.
- Divide the divisor by the remainder.
- Repeat this process till the remainder becomes zero.
- The last divisor is the GCD of the two.
- Solving by Example: Consider two numbers 188 and 423.
- The smaller number is 188.Therefore divide 423 by 188.
- Step 1



  - 47(Remainder) becomes the divisor and 188 (Divisor) becomes the dividend.
- Step 2



  - Since the remainder has become zero, 47 is the GCD of 188 and 423.

Algorithm Definition

Step 1: Start.

Step 2: Read two numbers for which GCD is to be found, say (a, b).

Step 3: Let a be the larger of the two numbers.

Step 4: Divide a by b.

Step 5: Get the remainder of the division operation.

Step 6: Divide the divisor of the previous division operation with the remainder.

Step 7: Repeat steps 4, 5, 6 till the remainder become zero.

Step 8: The divisor of the last division operation performed is the GCD of the two numbers.

Step 9: Stop.

To compute Greatest Common Divisor (GCD) of two numbers

```
#include <stdio.h>
 int gcd(int, int);
 int main()
{
   int a, b, result;
   printf("Enter the two numbers to find their GCD: ");
   scanf("%d%d", &a, &b);
   result = gcd(a, b);
   printf("The GCD of %d and %d is %d.\n", a, b, result);
}

int gcd(int a, int b)
{
   while(a!= b)
   {
      if(a > b)
      {
         return gcd(a - b, b);
      }
      else
      {
         return gcd(a, b - a);
      }
   }
   return a;
}
}
```

Test Whether a Number is Prime

- Problem Definition: To verify whether an integer is prime or not.
- Problem Analysis: Prime number is an integer which is exactly divisible by 1 and itself.
- For example, 17 is a prime number because 1 and 17 are the only two factors of 17.
- Generalizing this, a positive integer with just two factors is a prime number.
- Examples of prime numbers are 2, 3, 5, 7, 11, 13, ...
- To verify whether a number (say) is prime or not, divide the number by 2 to n - 1.
- If the remainder is zero in any of the cases, then the number is not prime (such numbers are called composite).
- Mathematically, it is also enough to divide the number from 2 to √n
- Solving by Example: Let us take two numbers 37 and 49.
- Consider 37 first. The square root of 37 is 6 (approximately).
- Now, divide 37 by 2, 3, 4, 5 and 6.
- None of the division operation gives a remainder of zero.
- Hence, 37 is a prime number. Now, let us consider 49.
- The square root of 49 is 7.So let us divide 49 by 2, 3, 4, 5, 6 and 7.
- As 49 is divisible by 7, it is not a prime numbers.

Algorithm Definition

Step 1: Begin.

Step 2: Read the number n which we want to verify whether prime or not.

Step 3: Initialize the value of the divisor i as.

Step 4: Repeat the following steps till i less than or equal to square root of n. (or) the remainder of the division operation is zero.
(I) Divide n by i.
(II) If remainder is zero, then prints the number is not prime and exit the process.

Step 5: If none of the remainders are zero, print n is a prime number.

Step 6: Stop.

To check if the given number is prime

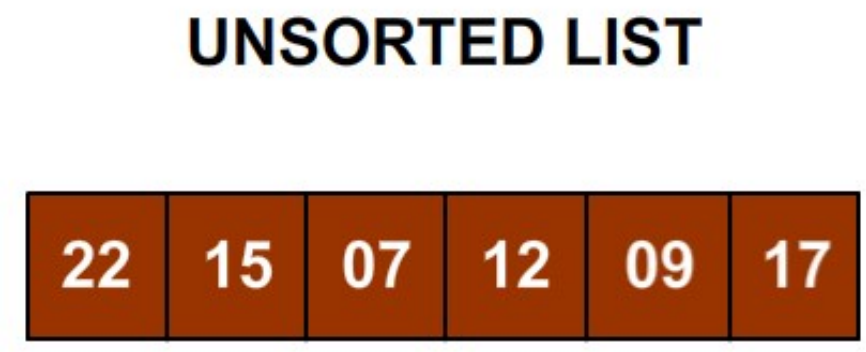
```
#include <stdio.h>
int main()
{
 int n, i, flag=0;
 printf("Enter a positive integer: ");
 scanf("%d", &n);
 for(i=2; i<=n/2; ++i)
 {
    if(n%i==0)
    {
       flag=1;
       break;
    }
 }
 if(flag==0)
    printf("%d is a prime number.",n);
 else
    printf("%d is not a prime number.",n);
 return 0;
}
```

Organize Numbers in Ascending Order

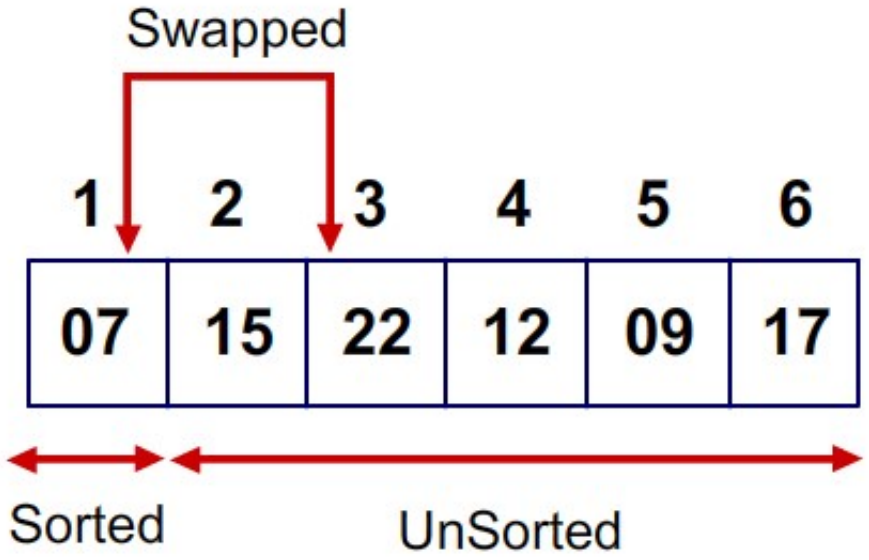
- Problem Definition: Organize a given set of numbers in ascending order.
- Problem Analysis: The process of organizing a given set of numbers in ascending/ descending order is called sorting.
- Though there are different flavors of sorting algorithms, selection sort is the one that is widely used.

Algoritham

- Selection sort involves identifying the smallest number in the array of numbers and placing it in the top of the list.
- Then the second smaller number is identified and placed second the list.
- This process continues for n number.
- Solving by Example: Consider the given array.

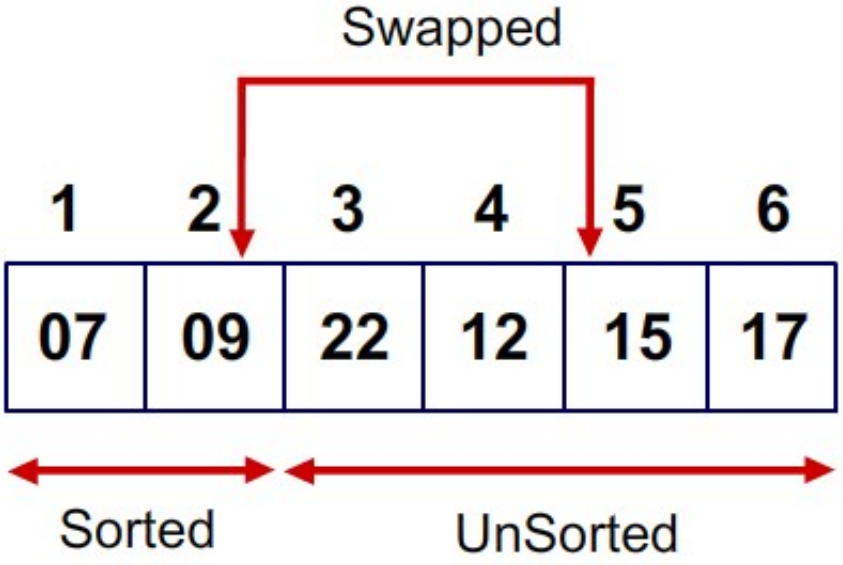


- Identify the smaller element in the array, the-element at position 3 is the smallest.
- Swap the elements at position 1 and position 3. The array becomes.



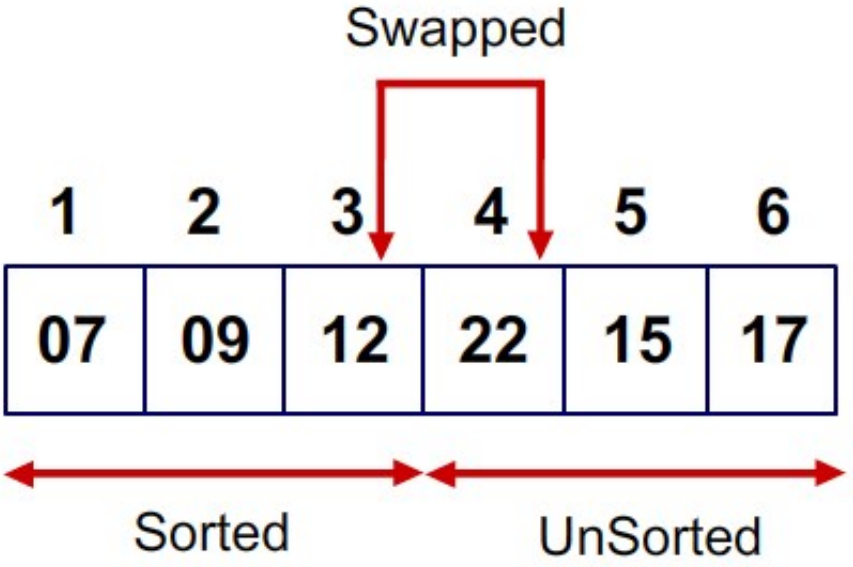
07 is identified as smaller

- Identify the second smallest number at position 5.
- Swap the elements at position 2 and position 5.

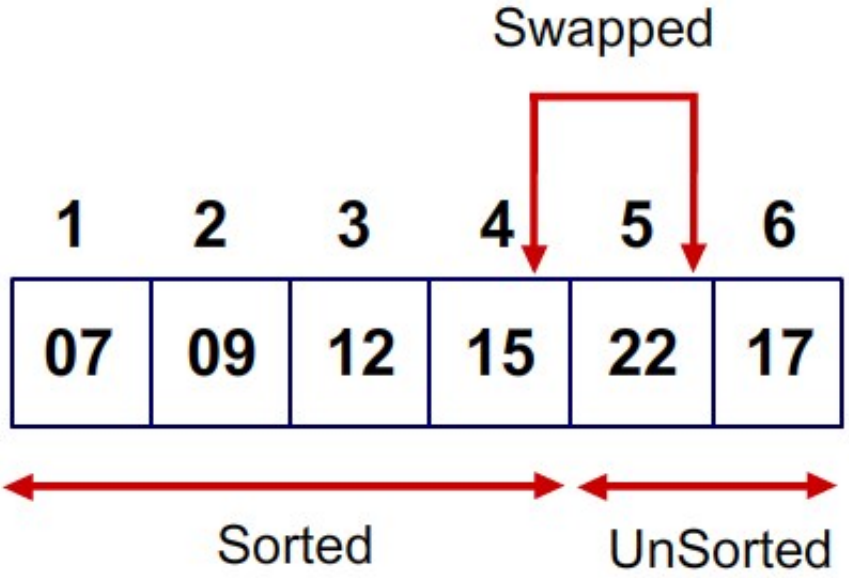


09 is identified as the second smaller

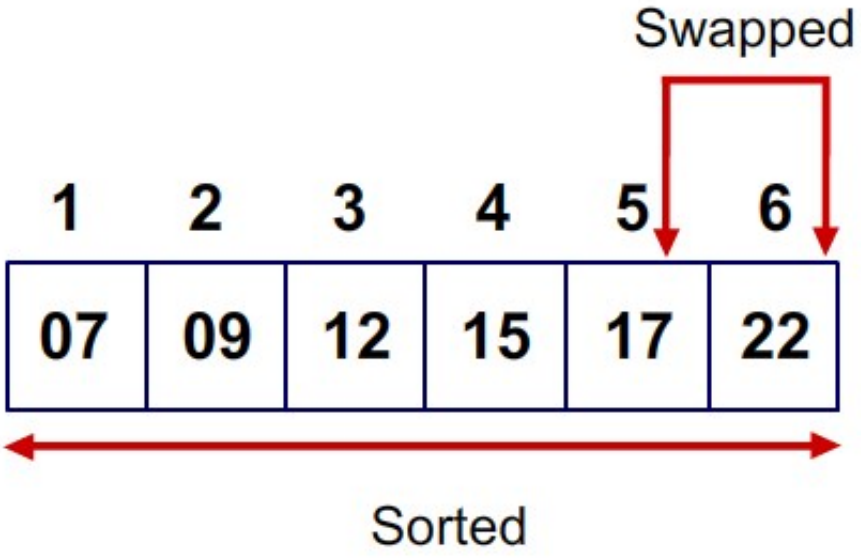
- Various snapshots of the sorting process is given below:



12 is identified as the third smallest



15 is identified as the fourth smallest



17 is identified as the fifth smallest

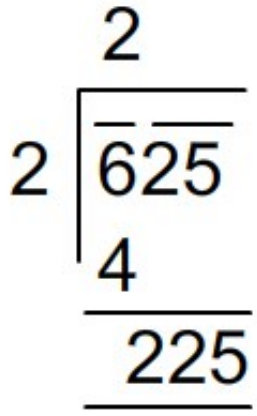
To organize numbers in ascending order

```
#include <stdio.h>
 void main()
{
   int i, j, a, n, number[30];
   printf("Enter the value of N \n");
   scanf("%d", &n);
   printf("Enter the numbers \n");
   for(i = 0; i < n; ++i)
      scanf("%d", &number[i]);
   for(i = 0; i < n; ++i)
   {
      for(j = i + 1; j < n; ++j)
      {
         if(number[i] > number[j])
         {
            a =  number[i];
            number[i] = number[j];
            number[j] = a;
         }
      }
   }
   printf("The numbers arranged in ascending order are given below \n");
   for(i = 0; i < n; ++i)
   printf("%d\n", number[i]);
}
```

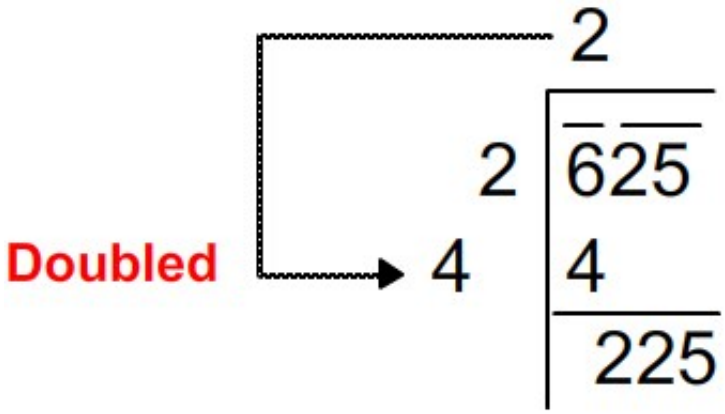
Find Square Root of a Number

- Problem Definition: Find the square root of an integer.
- Problem Analysis: Square root of a given number n is that natural number which when multiple by itself produces a product n.
- For example,
  - 2*2=4, so 2 is the square root of
  - 3*3=9, so 3 is the square root of
- So, if m is a square root of n, then it can be derived that m*m=n and it is symbolically denoted by √n = m,
- Solving by Example: There are two ways of finding the square root of a number.
- The first one is the trial-and-error method where we can zero upon the square root by multiplying different numbers to itself and deciding on the number which produces the nearest result.
- Another method is a bit complicated. For example, let us consider an integer 625.
- Now, place a bar over every pair of digits starting from the right.
- The numbers of bar's indicate the number of digits in the square root. We can divide 625 as

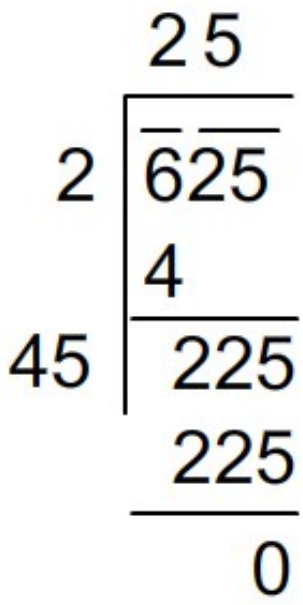
- Now, find the largest square root of the first pair which is equal to or less than the first pair.
- Take the square root of this number as divisor and derive the quotient.
- Subtract the first pair of numbers and bring the next period down.



- Now, double the quotient as it appears and leaves a blank digit to the right. This is the next divisor.



- The blank digit should be selected in such a way that multiplying the digit with the number formed as the divisor should be less than equal to or the dividend, For example, here 46 * 6 = 276 is greater than 225.
- Hence, we select 5 as the digit which satisfies our condition (45* 5=225). Put the digit to the quotient.



- Hence the square roots of 625.

To Find square root of a number

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

int main()
{
  printf("sqrt of 16 = %f\n", sqrt (16));
  printf("sqrt of 2 = %f\n", sqrt (2));
  return 0;
}
```

Factorial Computation

- To compute factorial of a given number:
  - Input: Given the Number to be computed => N.
  - Output: Factorial of N is Fact.
  - Step 1: Input Number whose Factorial is to be computed: N.
  - Step 2: Initialize Fact =1.
    - Assign N = M.
  - Step 3: Check if M is greater than 1.
    - Yes: Assign product of Fact and M to Fact.
    - Decrement M. Repeat Step 3.
    - No: Output Factorial of N is Fact.
  - Step 4: Stop.

To compute factorial of a given number

```
#include <stdio.h>
 int main()
{
 int c, n, fact = 1;
 printf("Enter a number to calculate it's factorial\n");
 scanf("%d", &n);
 for(c = 1; c <= n; c++)
 fact = fact * c;
 printf("Factorial of %d = %d\n", n, fact);
 return 0;
}
```

Fibonacci Sequence

- Algorithm: Fibonacci Series

Input: Range between which Fibonacci Series is to be generated. Low, High

Output: Fibonacci Series.

Step 1: Input Range between which Fibonacci Series is to be generated: Low, High.

Step 2: Assign A=0.
Assign B=1.

Step 3: If A is Greater Than or Equal to Low and
If B is lesser than or Equal to High
Yes: Output A.
No: Step 4

Step 4: If B is Greater Than or equal to Low and
B is lesser than or equal to High
Yes: Output B.
No: Step 5.

Step 5: Assign sum of A and B to C

Step 6: If C is Greater Than or Equal to Low and
C is lesser than or Equal to High
Yes: Output C.
No: Stop.

Step 7: Assign B = A, C = B,
Sum A and B to C
Repeat Step 6.

To Generate Fibonacci Series for given range

```
#include<stdio.h>
 int main()
{
  int n, first = 0, second = 1, next, c;
  printf("Enter the number of terms\n");
  scanf("%d",&n);
  printf("First %d terms of Fibonacci series are:-\n",n);
  for(c = 0; c < n; c++)
  {
    if(c <= 1)
      next = c;
    else
    {
      next = first + second;
      first = second;
      second = next;
    }
   printf("%d\n",next);
  }
 return 0;
}
```

Evaluate ‘sin x’ as Sum of a Series

- Input: Number whose sine function is to be computed: X.
- Number of iterations to be carried out: N.
- Output :Sine of the given Number
- Step 1: Input: Number whose sine function is to be computed: X.
  - Number of iterations to be carried out: N.
- Step 2: Initialize Sum = 0.
  - Den = 1.
- Step 3 : If N is Equal to One (1),
  - Yes: Output Sin(x) is X.
  - No: Step 4.
- Step 4: Initialize Flag =1.
  - Fact = 1
  - Assign X = Y.
- Step 5: Add Sum to flag multiplied by X divided by Fact and assign to sum: (Sum = (Flag * X)/Fact).
  - Assign the product of X and the Square of Y to X: (X = X * Y ^ 2).
  - Negate the sign of the flag.
  - Compute the product of Fact, Den and Den+1 and assign to Fact: (Fact = Fact*Den* (Den+1)).
  - Increment Den by 2.
  - Decrement N by 1.
  - Repeat Step 5 N times.
- Step 6: Output the Sum.

To Evaluate ‘sin x’ as Sum of a Series

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
void main()
{
   int n, x1;
   float accuracy, term, denominator, x, sinx, sinval;
   printf("Enter the value of x (in degrees) \n");
   scanf("%f", &x);
   x1 = x;
   /*  Converting degrees to radians */
   x = x * (3.142 / 180.0);
   sinval = sin(x);
   printf("Enter the accuracy for the result \n");
   scanf("%f", &accuracy);
   term = x;
   sinx = term;
   n = 1;
   do
   {
      denominator = 2 * n * (2 * n + 1);
      term = -term * x * x / denominator;
      sinx = sinx + term;
      n = n + 1;
   }
while (accuracy <= fabs(sinval - sinx));
printf("Sum of the sine series = %f \n", sinx);
printf("Using Library function sin(%d) = %f\n", x1, sin(x));
}
```

Reverse Order of Elements of an Array

Iterative way

- Initialize start and end indexes
  - start = 0, end = n-1
- In a loop, swap arr [start] with arr [end] and change start and end as follows.
  - start = start + 1, end = end – 1.

Recursion way

- Initialize start and end indexes
  - start = 0, end = n-1.
- Swap arr [start] with arr[end].
- Recursively call reverse for rest of the array.
- For example if a is an array of integers with three elements such that
  - a [0] = 1
  - a [1] = 2
  - a [2] = 3
- Then on reversing the array will be
  - a [0] = 3
  - a [1] = 2
  - a [0] = 1

To Reverse Order of Elements of an Array

```
#include <stdio.h>
int main()
{
  int n, c, d, a[100], b[100];
  printf("Enter the number of elements in array\n");
  scanf("%d", &n);
  printf("Enter the array elements\n");
  for(c = 0; c < n; c++)
  scanf("%d", &a[c]);
  for(c = n - 1, d = 0; c >= 0; c--, d++)
  b[d] = a[c];
  for (c = 0; c < n; c++)
  a[c] = b[c];
  printf("Reverse array is\n");
  for(c = 0; c < n; c++)
  printf("%d\n", a[c]);
  return 0;
}
```

Find Largest Number in Array

Problem Statement

- Design an algorithm to find maximum value among a set of N numbers.

Solution

- Algorithm Name: Search Max(A, N, R).
- Input : A is an array contains N (>=0) number of real numbers.
- Output: R will hold the index of the first maximum value in the the array A.
- Step 1: R = 0.
- Step 2: i = 1.
- Step 3: WHILE (i< N)
  IF (A[i] > A[R]) Then
  R = i
  EndIf
  i = i + 1
  EndWhile.
- Step 4: Return.

To Find Largest Number in Array

```
#include <stdio.h>
 int main()
{
 int array[100], maximum, size, c, location = 1;
 printf("Enter the number of elements in array\n");
 scanf("%d", &size);
 printf("Enter %d integers\n", size);
 for(c = 0; c < size; c++)
 scanf("%d", &array[c]);
maximum = array[0];
for (c = 1; c < size; c++)
 {
   if(array[c] > maximum)
   {
     maximum  = array[c];
     location = c+1;
   }
 }
 printf("Maximum element is present at location %d and it's value is %d.\n",
 location, maximum);
 return 0;
}
```

Print Elements of Upper Triangular Matrix

- Input: The Square Matrix of length N.
- Output: The Upper Triangle Matrix.
- Step 1: Input The Order of the Matrix: m x n.
- Step 2: Input The Elements of the Matrix A[I, J].
- Step 3: Initialize i, j.
- Step 4: Check if i< rows and increment i(i++).
- Step 5: Check if j< column and increment j(j++).
- Print a[i][j].
- Stop.

To Print Elements of Upper Triangular Matrix

```
#include<stdio.h>
int main()
{
int a[3][3],i,j;
float determinant=0;
printf("Enter the 9 elements of matrix: ");
for(i=0;i<3;i++)
for(j=0;j<3;j++)
scanf("%d",&a[i][j]);
printf("\nThe matrix is\n");
for(i=0;i<3;i++)
{
printf("\n");
for(j=0;j<3;j++)
printf("%d\t",a[i][j]);
}
printf("\nSetting zero in upper triangular matrix\n");
for(i=0;i<3;i++)
{
printf("\n");
for(j=0;j<3;j++)
if(i>=j)
printf("%d\t",a[i][j]);
else
printf("%d\t",0);
}
return 0;
}
```

Multiplication of Two Matrices

- Given two matrices A and B of orders m*n and p*q respectively.
- This algorithm multiplies the two matrices and stores the result in matrix C of order m*q I, J, K denotes array indices.

Algorithm of multiplication of two matrices

- Input: Two matrixes.
- Output: Output matrix C.
- Matrix-Multiply(A, B)
  - if columns [A] ≠ rows [B]
  - then error "incompatible dimensions"
  - else
  - for i =1 to rows [A]
  - for j = 1 to columns [B]
  - C[i, j] =0
  - for k = 1 to columns [A]
  - C[i, j] = C[i, j] + A[i, k] * B[k, j]
  - return C.

Algorithm Description

- To multiply two matrixes sufficient and necessary condition is "number of columns in matrix A = number of rows in matrix B".
- Loop for each row in matrix A.
- Loop for each columns in matrix B and initialize output matrix C to 0.
- This loop will run for each rows of matrix A.
- Loop for each columns in matrix A.
- Multiply A[i, k] to B[k, j] and add this value to C[i, j].
- Return output matrix C.

Evaluate a Polynomial

- The polynomial equation formula is

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

- Here is source code of the C program to evaluate the given polynomial equation.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define MAXSIZE 10
void main()
{
  int a[MAXSIZE];
  int i, N,power;
  float x, polySum;
  clrscr();
  printf("Enter the order of the polynomial\n");
  scanf("%d", &N);
  printf("Enter the value of x\n");
  scanf("%f", &x);
  /*Read the coefficients into an array*/
  printf("Enter %d coefficients\n",N+1);
  for (i=0;i <= N;i++)
  {
   scanf("%d",&a[i]);
  }
  polySum = a[0];
  for(i=1;i<= N;i++)
  {
   polySum = polySum * x + a[i];
  }
  power = N;
  /*power--;*/
  printf("Given polynomial is:\n");
  for (i=0;i<= N;i++)
  {
    if (power < 0)
    {
        break;
    }
```

```
    /* printing proper polynomial function*/
    if (a[i] > 0)
          printf(" + ");
    else if (a[i] < 0)
          printf(" - ");
    else
          printf (" ");
    printf("%dx^%d ",abs(a[i]),power--);
  }
  printf("\nSum of the polynomial = %6.2f\n",polySum);
}
```

UNIT - 3

Introduction to ‘C’ Language

History of C

- C was invented and first implemented by Dennis Ritchie on DEC PDP-11 that used the UNIX operating system.
- C is the result of a development process that started with an older language called BCPL.
- BCPL was developed by Martin Richards, and it influenced a language called B, which was invented by Ken Thompson.
- B led to development of C in the 1970.
- To alter this situation, ANSI established a committee in the beginning of 1983 to create a standard for C, which was implemented in 1987.

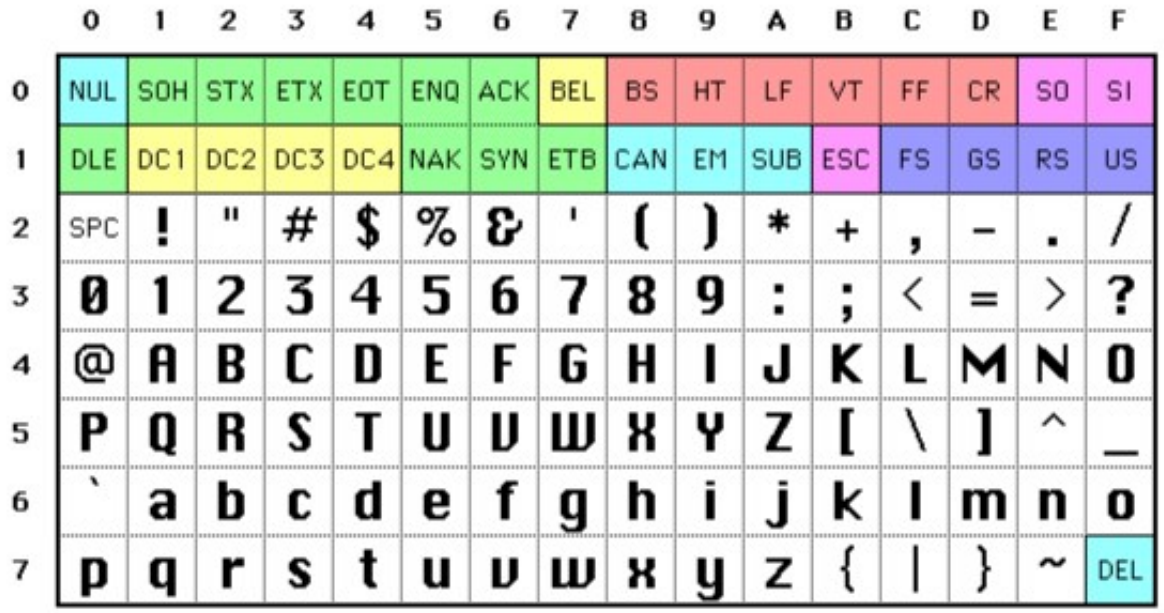
ASCII Code

- ASCII is an acronym for the American Standard Code for Information Interchange.
- ASCII is a code for representing English characters as numbers, with each letter assigned a number from 0 to 127.
- For example, the ASCII code for uppercase M is 77.
- Most computers use ASCII codes to represent text, which makes it possible to transfer data from one computer to another.
- Text files stored in ASCII format are sometimes called ASCII files.
- Text editors and word processors are usually capable of storing data in ASCII format, although ASCII format is not always the default storage format.
- Most data files, particularly if they contain numeric data, are not stored in ASCII format.
- Executable programs are never stored in ASCII format.
- The standard ASCII character set uses just 7 bits for each character.
- There are several larger character sets that use 8 bits, which gives them 128 additional characters.
- The extra characters are used to represent non-English characters, graphics symbols, and mathematical symbols.

Character Set: What is a character set?

- A character set is the mapping of characters to binary values.
- In 8 bit character sets, the values range from 0-255 and one character will be mapped to each of these values.
- This means that if the users terminal is set to support a particular character set; every time he press a certain key or a combination of keys a corresponding character will be invoked.

ASCII - Character Set



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

C Character Set

- These are the characters that C recognizes.
- Letters (upper case and lower case)
  - A B C D E … and so on.
  - a b c d e … and so on.
- Digits
  - 0 1 2 3 … and so on.
- Special Characters (punctuation etc), space (also known as blank)
  - ′ ″ ( ) * + - / : = ! & $; <> % ? , . ^ # @ ~ ‘ { } [ ] \ |

Variables and Identifiers

Variables

- A variable is a named data storage location in user computer's memory.
- By using a variable's name in the program, the users are, in effect, referring to the data stored there.
- For example:
  - Age = 10; implies that age hold the value 10.
  - Age = 20; now implies that age hold the value 20.
  - So from the above example age changed its value from 10 to 20.
  - Hence age can be referred to as a variable.

Identifier

- In C the names that are used to reference variables, functions, labels and various other user-defined objects are called identifiers.
- The length of an identifier in C can vary from one to several characters.
- In most cases the first character must be a letter or an underscore, and subsequent characters can be a letter, number or an underscore.
- Identifiers are used extensively in virtually all information processing systems.
- Naming entities makes it possible to refer to them, which is essential for any kind of symbolic processing.

Variable Names

- To use variables in the user C program, he must know how to create variable names.
- In C, variable names must adhere to the following rules:
  - The name can contain letters, digits, and the underscore character (_).
  - The first character of the name must be a letter. The underscore is also a legal first character, but its use is not recommended.
  - Case matters (that is, upper- and lowercase letters). Thus, the names count and Count refer to two different variables.
  - C keywords can't be used as variable names. A keyword is a word that is part of the C language.

Variable Name Legality

- The following list contains some examples of legal and illegal C variable names:

Variable Name	Legality
Percent	Legal
y2x5_fg7h	Legal
annual_profit	Legal
_1990_tax	Legal but not advised.
saving#account	Illegal: Contain the illegal character #.
double	Illegal: Is a C Keyword.
9winter	Illegal: First character is a digit.

- Using an underscore to separate words in a variable name makes it easy to interpret.
- The second style is called camel notation.
- Instead of using spaces, the first letter of each word is capitalized.
- Instead of interest -rate, the variable would be named Interest Rate.
- Camel notation is gaining popularity, because it′s easier to type a capital letter than an underscore.
- We use the underscore because it′s easier for most people to read.
- User should decide which style, he wants to adopt.

Built in Data Types

Data type

- Two types of built-in data types:
  - Fundamental data types (int, char, double, float, void, pointer).
  - Derived data types (array, string, structure).
- It is a way of representing data storage formats.
- There are five basic data types in C.
- Those are char for character, int for integer, float for floating point, double - double precision floating point, and void.
- Values of type char are, in theory, restricted to the defined ASCII characters.
- The range of types float and double is usually given in digits of precision.
- The magnitude of type float and double depend upon the method used to represent the floating point numbers.
- Type void is used to explicitly declare a function as returning no value.

Fundamental Data Types

- void – used to denote the type with no values.
- void cannot be used to define a variable.
- void, in C, is a type that has no size.
- Thus, if one was to declare a variable of type "void", the compiler would not know how much memory to allocate for it.
- Therefore void cannot be used to define a variable.
- int – used to denote an integer type.
- char – used to denote a character type.
- float, double – used to denote a floating point type.
- int *, float *, char * – used to denote a pointer type, which is a memory address type.

Derived data types

- Array – a finite sequence (or table) of variables of the same data type.
- String – an array of character variables.

- Structure – a collection of related variables of the same and/or different data types.
- The structure is called a record and the variables in the record are called members or fields.

Numeric data Types

- C provides several different types of numeric variables.
- The users need different types of variables because different numeric values have varying memory storage requirements and differ in the ease with which certain mathematical operations can be performed on them.
- Small integers (for example, 1, 199, and -8) require less memory to store, and the user computer can perform mathematical operations (addition, multiplication, and so on) with such numbers very quickly.
- In contrast, large integers and floating-point values (123,000,000 or 0.000000871256, for example) require more storage space and more time for mathematical operations.
- By using the appropriate variable types, the users ensure that the program runs as efficiently as possible.
- Listed below are the numeric data types in c.

Type	Bytes	Values
int	2 or 4	-32, 768 to 32, 767
unsigned int	2 or 4	0 to 65, 535
signed int	2 or 4	-32, 767 to 32, 767
short int	2	-32, 767 to 32, 767
unsigned short int	2	0 to 65, 535
signed short int	2	-32, 767 to 32, 767
long int	4	-2, 147, 483, 647 to 2, 147, 483, 647
signed long int	4	-2, 147, 483, 647 to 2, 147, 483, 647
unsigned long int	4	0 to 4, 294, 967, 294

Variable Definition

- A variable is a way of referring to a memory location used in a computer program.
- This memory location holds values- perhaps numbers or text or more complicated types of data like a payroll record.

Variable Declarations

- Before the users can use a variable in a C program, it must be declared.
- A variable declaration tells the compiler the name and type of a variable and optionally initializes the variable to a specific value.
- If the program attempts to use a variable that hasn′t been declared, the compiler generates an error message.
- A variable declaration has the following form:
  - typename varname;
    - typename specifies the variable type and must be one of the keywords.
- From the declarations, varname is the variable name, which must follow the rules mentioned earlier.
- User can declare multiple variables of the same type on one line by separating the variable names with commas:
  - int count, number, start; /* three integer variables */
  - float percent, total; /* two float variables */
  - char firstname; /* character type variable */

Character Variables

- A character is a single letter, numeral, punctuation mark, or other such symbol.
- A string is any sequence of characters.
- Strings are used to hold text data, which is comprised of letters, numerals, punctuation marks, and other symbols.
- Using Character Variables, like other variable the users must declare chars before using them, and he can initialize them at the time of declaration.
- Here are some examples of character variables:
  - char a, b, c; /* Declare three uninitialized char variables */
  - char code = ‘x’; /* Declare the char variable named code and store the character x there */

- code = ‘!’; /* Store ! in the variable named code */

Integer, Float and Double declarations

- To declare a variable as integer, follow the below syntax:
  - int variable_name;
    - int is the type of the variable named variable_name.
    - ‘int’ denotes integer type.
- A float is a single-precision floating point value.
- To declare a variable as float, follow the below syntax:
  - float variable_name;
- A double is a double-precision floating point value.
- To declare a variable as double, follow the below syntax:
  - double variable_name;
- Examples:
  - int i, j, k;
  - float f, salary;
  - double d;

Expression and Operators

- The symbols which are used to perform logical and mathematical operations in a C program are called C operators.
- These C operators join individual constants and variables to form expressions.
- Operators, functions, constants and variables are combined together to form expressions.
- Consider the expression A + B * 5.
- Where, +, * are operators, A, B are variables, 5 is constant and A + B * 5 is an expression.

Types of C operators

- C language offers many types of operators.
- They are,
  - Arithmetic operators.
  - Assignment operators.
  - Relational operators.
  - Logical operators.
  - Bit wise operators.
  - Conditional operators (ternary operators).
  - Increment/decrement operators.
  - Special operators.

Arithmetic Operators in C

- C Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in C programs.

S.no	Arithmetic Operators	Operation	Example
1	+	Addition	A+B
2	-	Subtraction	A-B
3	*	Multiplication	A*B
4	/	Division	A/B
5	%	Modulus	A%B

Example program for C arithmetic operators

- In this example program, two values "40" and "20" are used to perform arithmetic operations such as addition, subtraction, multiplication, division, modulus and output is displayed for each operation.

```
#include <stdio.h>
int main()
{
   int a=40, b=20, add, sub, mul, div, mod;
   add = a+b;
   sub = a-b;
   mul = a*b;
   div = a/b;
   mod = a%b;
   printf("Addition of a, b is: %d\n", add);
   printf("Subtraction of a, b is: %d\n", sub);
   printf("Multiplication of a, b is: %d\n", mul);
   printf("Division of a, b is: %d\n", div);
   printf("Modulus of a, b is: %d\n", mod);
}
```

Output:

```
Addition of a, b is: 60
Subtraction of a, b is: 20
Multiplication of a, b is: 800
Division of a, b is: 2
Modulus of a, b is: 0
```

Assignment operators

- In C programs, values for the variables are assigned using assignment operators.
- For example, if the value "10" is to be assigned for the variable "sum", it can be assigned as "sum = 10;".
- Other assignment operators in C language are given below.

Operators		Example	Explanation
Simple assignment operator	=	sum=10	10 is assigned to variable sum
	+=	sum+=10	This is same as sum = sum+10

Compound assignment operators	-=	sum-=10	This is same as sum = sum-10
	*=	sum*=10	This is same as sum = sum*10
	/=	sum/=10	This is same as sum = sum/10
	%=	sum%=10	This is same as sum = sum%10
	&=	sum&=10	This is same as sum = sum&10
	^=	sum^=10	This is same as sum = sum^10

Example program for C assignment operators

- In this program, values from 0 – 9 are summed up and total “45″ is displayed as output.
- Assignment operators such as “=” and “+=” are used in this program to assign the values and to sum up the values.

```
# include <stdio.h>
int main()
{
   int Total=0,i;
   for(i=0;i<10;i++)
   {
      Total+=i; // This is same as Total = Total+i
   }
   printf("Total = %d", Total);
}
```

Output:

```
Total = 45
```

Relational operators

- Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables in a C program.

S.no	Operators	Example	Description
1	>	x > y	x is greater than y
2	<	x < y	x is less than y
3	>=	x >= y	x is greater than or equal to y

4	<=	x <= y	x is less than or equal to y
5	==	x == y	x is equal to y
6	!=	x != y	x is not equal to y

Example program for relational operators in C

- In this program, relational operator (==) is used to compare 2 values whether they are equal are not.
- If both values are equal, output is displayed as " values are equal".
- Else, output is displayed as "values are not equal".
- Note : double equal sign (==) should be used to compare 2 values.
- We should not single equal sign (=).

```
#include <stdio.h>
int main()
{
   int m=40, n=20;
   if(m == n)
   {
      printf("m and n are equal");
   }
   else
   {
      printf("m and n are not equal");
   }
}
```

Output:

```
m and n are not equal
```

Logical operators

- These operators are used to perform logical operations on the given expressions.
- There are 3 logical operators in C language.
- They are, logical AND (&&), logical OR (||) and logical NOT (!).

S.no	Operators	Name	Example	Description

1	&&	logical AND	(x>5)&&(y<5)	It returns true when both conditions are true.
2	||	logical OR	(x>=10)|| (y>=10)	It returns true when at-least one of the condition is true.
3	!	logical NOT	!((x>5)&& (y<5))	It reverses the state of the operand “((x>5) && (y<5))” If “((x>5) && (y<5))” is true, logical NOT operator makes it false

Example program for logical operators in C

- In this program, operators (&&, || and !) are used to perform logical operations on the given expressions.
- && operator
  - “if clause” becomes true only when both conditions (m>n and m! =0) is true.
  - Else, it becomes false.
- || Operator
  - “if clause” becomes true when any one of the condition (o>p || p!=20) is true.
  - It becomes false when none of the condition is true.
- ! Operator
  - It is used to reverses the state of the operand.
  - If the conditions (m>n && m!=0) is true, true (1) is returned.
  - This value is inverted by “!” operator.
  - So, “! (m>n and m! =0)” returns false (0).

```
#include <stdio.h>
int main()
{
    int m=40, n=20;
    int o=20, p=30;
    if(m>n && m!=0)
    {
        printf("&& Operator:  Both conditions are true\n");
    }
    if(o>p || p!=20)
    {
        printf("|| Operator: Only one condition is true\n");
    }
    if(!(m>n && m!=0))
    {
       printf("! Operator: Both conditions are true\n");
    }
    else
    {
       printf("! Operator: Both conditions are true. " \
            "But, status is inverted as   false\n");
    }
}
```

Output

```
&& Operator:  Both conditions are true
|| Operator: Only one condition is true
! Operator: Both conditions are true. But, status is inverted as false
```

Bit wise operators

- These operators are used to perform bit operations.
- Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.
- Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (right shift).

Truth table for bit wise operation

x	y	x|y	x & y	x ^ y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1

1	1	1	1	0

Bit wise operators

Operator symbol	Operator name
&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT
^	XOR
<<	Left Shift
>>	Right Shift

- Consider x=40 and y=80.
- Binary form of these values are given below.
  - x = 00101000.
  - y= 01010000.
- All bit wise operations for x and y are given below.
  - x&y = 00000000 (binary) = 0 (decimal).
  - x|y = 01111000 (binary) = 120 (decimal).
  - ~x = 11111111111111111111111111111111111111111111111111111111010111 = -41 (decimal).
  - x^y = 01111000 (binary) = 120 (decimal).
  - x << 1 = 01010000 (binary) = 80 (decimal).
  - x >> 1 = 00010100 (binary) = 20 (decimal).
- Note:
  - Bit wise NOT : Value of 40 in binary is 00000000000000000000000000000000000000000000000000000000101000.
  - So, all 0′s are converted into 1′s in bit wise NOT operation.
  - Bit wise left shift and right shift : In left shift operation “x << 1 “, 1 means that the bits will be left shifted by one place.

    If we use it as “x << 2 “, then, it means that the bits will be left shifted by 2 places.

Example program for bit wise operators in C

- In this example program, bit wise operations are performed as shown above and output is displayed in decimal format.

```
#include <stdio.h>
int main()
{
   int m=40, n=80, AND_opr, OR_opr, XOR_opr, NOT_opr;
   AND_opr = (m&n);
   OR_opr = (m|n);
   NOT_opr = (~m);
   XOR_opr = (m^n);
   printf("AND_opr value = %d\n", AND_opr);
   printf("OR_opr value = %d\n", OR_opr);
   printf("NOT_opr value = %d\n", NOT_opr);
   printf("XOR_opr value = %d\n", XOR_opr);
   printf("left_shift value = %d\n", m << 1);
   printf("right_shift value = %d\n", m >> 1);
}
```

Output:

```
AND_opr value = 0
OR_opr value = 120
NOT_opr value = -41
XOR_opr value = 120
left_shift value = 80
right_shift value = 20
```

Conditional or ternary operators

- Conditional operators return one value if condition is true and returns another value is condition is false.
- This operator is also called as ternary operator.
  - Syntax : (Condition? true_value: false_value);
  - Example : (A > 100 ? 0 : 1);
- In above example, if A is greater than 100, 0 is returned else 1 is returned.
- This is equal to if else conditional statements.

Example program for conditional/ternary operators in C

```
#include <stdio.h>
int main()
{
        int x=1, y;
        y = (x ==1 ? 2 : 0);
        printf("x value is %d\n", x);
        printf("y value is %d", y);
}
```

Output:

x value is 1
y value is 2

Increment/decrement Operators

- Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.
- Syntax:
  - Increment operator : ++var_name; (or) var_name++;
  - Decrement operator : – - var_name; (or) var_name – -;
- Example:
  - Increment operator : ++ i ; i ++ ;
  - Decrement operator: – - i ; i – - ;

Example program for increment operators in C

- In this program, value of “i” is incremented one by one from 1 up to 9 using “i++” operator and output is displayed as “1 2 3 4 5 6 7 8 9”.

```
#include <stdio.h>
int main()
{
    int i=1;
    while(i<10)
    {
       printf("%d ",i);
       i++;
    }
}
```

Output:

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

Example program for decrement operators in C

- In this program, value of “I” is decremented one by one from 20 up to 11 using “i–” operator and output is displayed as “20 19 18 17 16 15 14 13 12 11”.

```
#include <stdio.h>
int main()
{
   int i=20;
   while(i>10)
   {
       printf("%d ",i);
       i--;
   }
}
```

Output:

```
20 19 18 17 16 15 14 13 12 11
```

Special Operators in C

- Below are some of special operators that C language offers.

S.no	Operators	Description
1	&	This is used to get the address of the variable. Example : &a will give address of a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.

3	Sizeof ()	This gives the sizeof the variable. Example : sizeof(char) will give us 1.

Example program for & and * operators in C

- In this program, “&” symbol is used to get the address of the variable and “*” symbol is used to get the value of the variable that the pointer is pointing to.
- Please refer C – pointer topic to know more about pointers.

```
#include <stdio.h>

int main()
{
      int *ptr, q;
      q = 50;
      /* address of q is assigned to ptr      */
      ptr = &q;
      /* display q's value using ptr variable */
      printf("%d", *ptr);
      return 0;
}
```

Output:

```
50
```

Example program for sizeof() operator in C

- sizeof() operator is used to find the memory space allocated for each C data types.

```
#include <stdio.h>
#include <limits.h>
int main()
{
  int a;
  char b;
  float c;
  double d;
  printf("Storage size for int data type:%d \n", sizeof(a));
  printf("Storage size for char data type:%d \n", sizeof(b));
  printf("Storage size for float data type:%d \n", sizeof(c));
  printf("Storage size for double data type:%d\n", sizeof(d));
  return 0;
}
```

Output:

```
Storage size for int data type: 4
Storage size for char data type: 1
Storage size for float data type: 4
Storage size for double data type: 8
```

Constants and Literals

- Like a variable, a constant is a data storage location used by the users program.
- Unlike a variable, the value stored in a constant can′t be changed during program execution.
- C has two types of constants, each with its own specific uses.

Literal Constants

- 20 and ‘R’ arethe examples for literal constant:
  - int count = 20;
  - char name= ‘R’;

Symbolic Constants

- A symbolic constant is a constant that is represented by a name (symbol) in the program.
- Like a literal constant, a symbolic constant can′t change.
- Whenever the User needs the constant′s value in the program, he can use its name as would use a variable name.
- The actual value of the symbolic constant needs to be entered only once, when it is first defined.
- Symbolic constants have two significant advantages over literal constants.
- Suppose that the user′s writing a program that performs a variety of geometrical calculations.
- The program frequently needs the value of π (PI) - (3.14159) for its calculations.
- The value of π is constant.
- For example:
  - To calculate the circumference and area of a circle with a known radius, the user could write the following code:
    - Circumference = 3.14159 * (2 * radius);
    - Area = 3.14159 * (radius) * (radius);
- The asterisk (*) is C′s multiplication operator.
- Thus, the first of these statements means Multiply 2 times the value stored in the variable radius, and then multiply the result by 3.14159.
- Finally, assign the result to the variable named circumference.
- If, however, the user define a symbolic constant with the name PI (π) and the value 3.14, he could write the following code:

- Circumference = PI * (2 * radius);
- Area = PI * (radius) * (radius);

- The second advantage of symbolic constants becomes apparent when the user’s need to change a constant.
- Continuing with the preceding example, The user might decide that for greater accuracy the program needs to use a value of PI with more decimal places:
  - 3.14159 Rather than 3.14.
  - If the user had used literal constants for PI, he would have to go through the source code and change each occurrence of the value from 3.14 to 3.14159.
  - With a symbolic constant, he needs to make a change only in the place where the constant is defined.
  - The rest of code would not need to be changed.

Defining symbolic constants

- C has two methods for defining a symbolic constant:
  - the #define directive and
  - the const keyword.
- The first way to define a symbolic constant is the #define directive is used as follows:
  - #define CONSTNAME literal
- This creates a constant named CONSTNAME with the value of literal. literal represents a literal constant, as described earlier.
- CONSTNAME follows the same rules described earlier for variable names.
- By convention, the names of symbolic constants are uppercase.
- This makes them easy to distinguish from variable names, which by convention are lowercase.
- For the previous example, the required #define directive for a constant called PI would be
  - #define PI 3.14159.
- Note that #define lines don′t end with a semicolon(;).
- #define statements can be placed anywhere in your source code, but the defined constant is in effect only for the portions of the source code that follow the #define directive.
- Most commonly, programmers group all #define statements together, near beginning of the file and before the start of the main() function.

How a #define works?

- The precise action of the #define directive is to instruct the compiler as follows:
- In the source code, replace CONSTNAME with literal.
- The effect is exactly the same as if the user had used the editor to go through the source code and make the changes manually.
- For example, #define PI 3.14159.
- /* the users have defined a constant for PI. */
- #define PIPE 100.

Defining Constants with the const keyword

- The second way to define a symbolic constant is with the const keyword.
- Const is a modifier that can be applied to any variable declaration.
  - const int a=10;
- A variable declared to be const can′t be modified during program execution--only initialized at the time of declaration.
- Here are some examples:
  - const int count = 100;
  - const long debt = 12000000, float tax_rate = 0.21;
- Const affects all variables on the declaration line. In the last line, debt and tax_rate are symbolic constants.

Literals

- The constants refer to fixed values that the program may not alter during its execution.
- These fixed values are also called literals.
- Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal.
- There are also enumeration constants as well.

Integer literals

- An integer literal can be a decimal, octal, or hexadecimal constant.
- A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

- An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively.
- The suffix can be uppercase or lowercase and can be in any order.

Floating-point literals

- A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part.
- The user can represent floating point literals either in decimal form or exponential form.
- While representing using decimal form, the user must include the decimal point, the exponent, or both and while representing using exponential form; he must include the integer part, the fractional part, or both.
- The signed exponent is introduced by e or E.

Character literals

- Character literals are enclosed in single quotes, e.g., ‘x’ and can be stored in a simple variable of char type.
- A character literal can be a plain character (e.g., ‘x’), an escape sequence (e.g., ‘\t’), or a universal character (e.g., ‘\u02C0’).
- There are certain characters in C when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t).
- Here, the user have a list of some of such escape sequence codes:

String literals

- String literals or constants are enclosed in double quotes "".
- A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.
- The users can break a long line into multiple lines using string literals and separating those using whitespaces.

Simple Assignments Statement

- A statement is a complete direction instructing the computer to carry out some task.
- In C, statements are usually written one per line, although some statements span multiple lines.
- C statements always end with a semicolon (except for #define and #include).
- Statements and White Space:
  - The term white space refers to spaces, tabs and blank lines in the user source code.
  - The C compiler isn′t sensitive to white space.
  - When the compiler reads a statement in the user source code, it looks for the characters in the statement and for the terminating semicolon, but it ignores white space.
- Thus, the statement:
  - x=2+3; is equivalent to x = 2 + 3; with white spaces.
  - It is also equivalent to as follows:
    - x =
    - 2
    - +
    - 3;
- This gives the user a great deal of flexibility in formatting the source code.
- Statements should be entered one per line with a standardized scheme for spacing around variables and operators.

Basic Input Output Statements

- The basic input/output functions are
  - getchar,
  - putchar,
  - gets,
  - puts,
  - scanf and
  - printf.
- The first two functions, getchar and putchar, are used to transfer single characters.
- The next function gets and puts are used to input and output strings, and the last two functions, scanf and printf, permit the transfer of single characters, numerical values and strings.

getchar() Function

- getchar( ) function is used to read one character at a time from the key board.
- Syntax ch = getchar( ); where ch is a char Var.
- Ex:

```
main( )
{
char ch;
printf("Enter a char");
ch = getchar ( );
printf("ch =%c", ch);
}
```

Output

```
Enter a char M
M
ch = M
```

- When this function is executed, the computer will wait for a key to be pressed and assigns the value to the variable when the “enter” key pressed.

putchar() Function

- putchar( ) function is used to display one character at a time on the monitor.
- Syntax: putchar (ch);
- EX:

- char ch = „M‟
  putchar(ch);

- The Computer display the value char of variable ‘ch’ i.e M on the Screen.

gets() Function

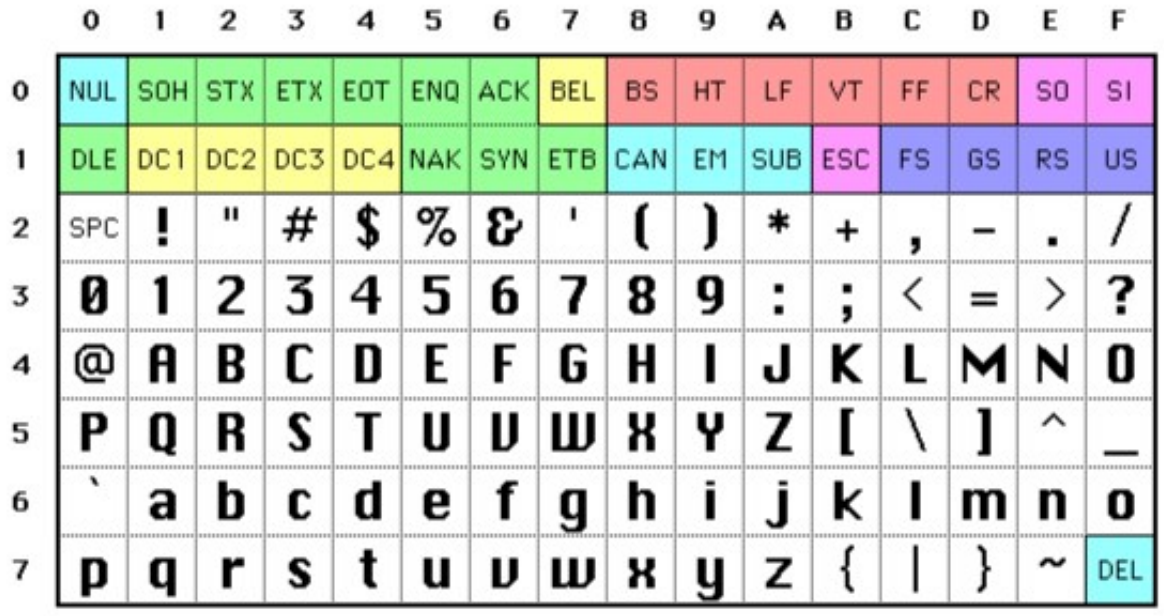
- gets( ) function is used to read a string of characters including white spaces.
- Note that white spaces in a string cannot be read using scanf( ) with %s format specifier.
- Syntax: gets (S); where ‘S’ is a char string variable.
- Ex:

  - char S[20];
    gets (S);

- When this function is executed the computer waits for the string to be entered.

Puts() Function

- puts() is a function used to display strings on screen.
- For example,



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

scanf()

- Just as most programs need to output data to the screen, they also need to input data from the keyboard.
- The most flexible way the program can read numeric data from the keyboard is by using the scanf() library function.
- The scanf() function reads data from the keyboard according to a specified format and assigns

the input data to one or more program variables.

- For example:
  - The statement reads a decimal integer from the keyboard and assigns it to the integer variable x as shown below:
    - scanf("%d", &x);
- The ‘%’ indicates that the conversion specification follows:
  - The ‘d’ represents the data type and indicates that the number should be read as a integer.
  - The ‘&’ is ‘C’ unary operator that gets the memory address of the variable following it.
  - The user will read more about this operator and its associated operator ‘*’, in the section ‘Pointers’ in the course.
  - Likewise, the following statement reads a floating-point value from the keyboard and assigns it to the variable rate:
    - scanf("%f", &rate);
  - The ‘f’ represents the data type and indicates that the number should be read as a float.

printf()

- The printf() function, part of the standard C library, is perhaps the most versatile way for a program to display data on-screen.
- Printing a text message on-screen is simple.
- Call the printf() function, passing the desired message enclosed in double quotation marks.
- For example, to display an error that has occurred! on-screen, the user write the following:
  - printf("An error that has occurred!");
- In addition to text messages, however, he frequently needs to display the value of program variables.
- This is a little more complicated than displaying only a message.
- It accepts a string parameter (called the format string), which specifies a method for rendering a number of other parameters (of which there typically may be arbitrarily many, of a variety of types) into a string.
- For example, suppose the user's want to display the value of the numeric variable x on-screen, along with some identifying text.
- Furthermore, he wants the information to start at the beginning of a new line.
- The printf() function as shown below:

- printf("\nThe value of x is %d", x);
- \n represents a new line character.

- The resulting screen display, assuming that the value of x is 12, would display the following:
  - The value of x is 12.
  - In this example, two arguments are passed to printf().
  - The first argument is enclosed in double quotation marks and is called the format string.
  - The second argument is the name of the variable (x) containing the value to be printed.

Simple ‘C’ programs

- The ‘Hello World’ introduction
  - The best way to learn a computer language is to start writing short programs that work and then gradually add complexity.
  - The traditional first C program prints out "hello, world" and looks something like this:

```
#include <stdio.h>
int main()
{
  /* my first program in C */
  printf("Hello, World! \n");
  return 0;
}
```

Use of the gets() Function

```
#include <stdio.h>
int main()
{
  char str[50];
  printf("Enter a string: ");
  gets(str);
  printf("You entered: %s", str);
  return(0);
}
```

To Multiply Two Integers

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a,b,c;
   a=5;
   b=10;
   c=mul(a,b);
   printf("Multiplication of %d and %d is %d",a,b,c);
}
```

Frequently used Escape Sequences

```
/* Testing the escape sequences */
#include <stdio.h>
 int main(void)
{
printf("Testing the escape sequences:\n");
printf("----------------------------\n");
 /* 3 times audible tone */
printf("The audible bell ---> \\a \a\a\a\n");
printf("The backspace ---> \\b___ \bTesting\n");
 /* printer must be attached...*/
printf("The formfeed, printer ---> \\f \fTest\n");
printf("The newline ---> \\n \n\n");
printf("The carriage return ---> \\r \rTesting\r\n");
printf("The horizontal tab ---> \\t \tTesting\t\n");
printf("The vertical tab ---> \\v Testing\v\n");
printf("The backslash ---> \\ \\Testing\\\n");
printf("The single quote ---> \' \'Testing\'\'\'\n");
printf("The double quote ---> \" \"Testing\"\"\n");
printf("Some might not work isn't it?\n");
return 0;
}
```

UNIT - 4

Conditional Statements and Loops

Decision Making within a Program - What is Decision making?

- Decision making is the thoughtful consideration and selection of a course of action from among available alternatives in order to produce a desired result.
- Most control statements in any computer language, including C, rely upon a conditional test that determines a course of action.
- The conditional test either evaluates to a true or a false.
- The concept of evaluating and obtaining a result is referred to as decision making in a programming language.
- "True" is considered the same as "yes," which is also considered the same as 1.
- "False" is considered the same as "no," which is considered the same as 0.

Decision Making and Conditions

- Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.
- Following is the general form of a typical decision making structure found in most of the programming languages.
- C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.
- C programming language provides following types of decision making statements.
- Decision is a word which is normally taken in a moment where one is in a position to select one option from the available options which are obviously more than one.
- While writing programs this concept is always going to play an important role.
- We will have to provide the capabilities to our program so that it can take decisions on its own depending upon the various possible inputs from the user.
- When you are going to write some program it will be always necessary that you will have to put some code which has to be executed only when a specific condition is satisfied.
- These conditions are evaluated by the computer itself when the input is given by the user.
- In C language there are various methods which can be used to select an appropriate set of statements depending upon the user's input.
- On counting them, totally there are FOUR different ways to take decisions which are as follows :
  - if Statement.

- if-else Statement.
- Conditional Operators.
- Switch Statement.

The Relational and Logical Operator

- The relational operators are used to compare expressions, asking questions such as, "Is y greater than 100?" or "Is z equal to 0?"
- An expression containing a relational operator evaluates to either true (1) or false (0).
- "True" is considered the same as "yes," which is also considered the same as 1.
- "False" is considered the same as "no," which is considered the same as 0.
- An expression created using a relational operator forms what is known as a relational expression or a condition.
- C's relational operators are as follows.

Operator	Meaning
<	Less than
< =	Less than or equal to
>	Greater than
> =	Greater than or equal to
==	Equal to
!=	Not equal to

Logical Operators

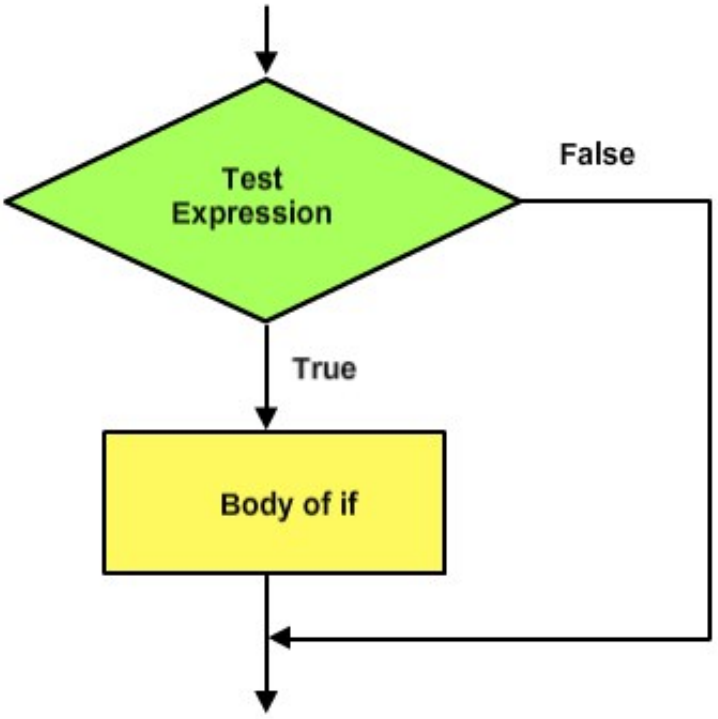
- Logical operators let the user's combine two or more relational expressions into a single expression that evaluates to either true or false.
- Logical operators consists of AND, OR and NOT as shown in the table below.
- An expression containing a Logical operator evaluates to either true (1) or false (0).
- "True" is considered the same as "yes," which is also considered the same as 1.
- "False" is considered the same as "no," which is considered the same as 0.

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

Control Statement

If statement

- A program control statement modifies the order of statement execution.
- Program control statements can cause other program statements to execute multiple times or not to execute at all, depending on the circumstances.
- The ‘if’ statement is one of the C's program control statements.
- In its basic form, the ‘if’ statement evaluates an expression and directs program execution depending on the result of the evaluation.



- If expression evaluates to true, statement is executed.
- If statement evaluates to false, statement is not executed.
- In either case, execution then passes to whatever code follows the ‘if’ statement.
- The execution of statement depends on the result of expression.
- The line if (expression) and the line statement; are considered to comprise the complete if statement; they are not separate statements.
- An ‘if’ statement can control the execution of multiple statements through the use of a compound statement, or block.
- A block is a group of two or more statements enclosed in braces.
- A block can be used anywhere a single statement can be used.

- Listed below is the syntax for if statement:
  - This is the ‘if’ statement in its simplest form.
  - If expression is true, statement1 is executed. If expression is not true, statement1 is ignored.

If statement Syntax

```
..
if (condition)
{
    //Block of C statements here
    //The above statements will only execute if the condition is true
}
..
```

If statement: Example

```
#include <stdio.h>
int main()
{
   int x = 20;
   int y = 22;
   if (x<y)
   {
      printf("Variable x is less than y");
   }
   return 0;
}
```

Output:

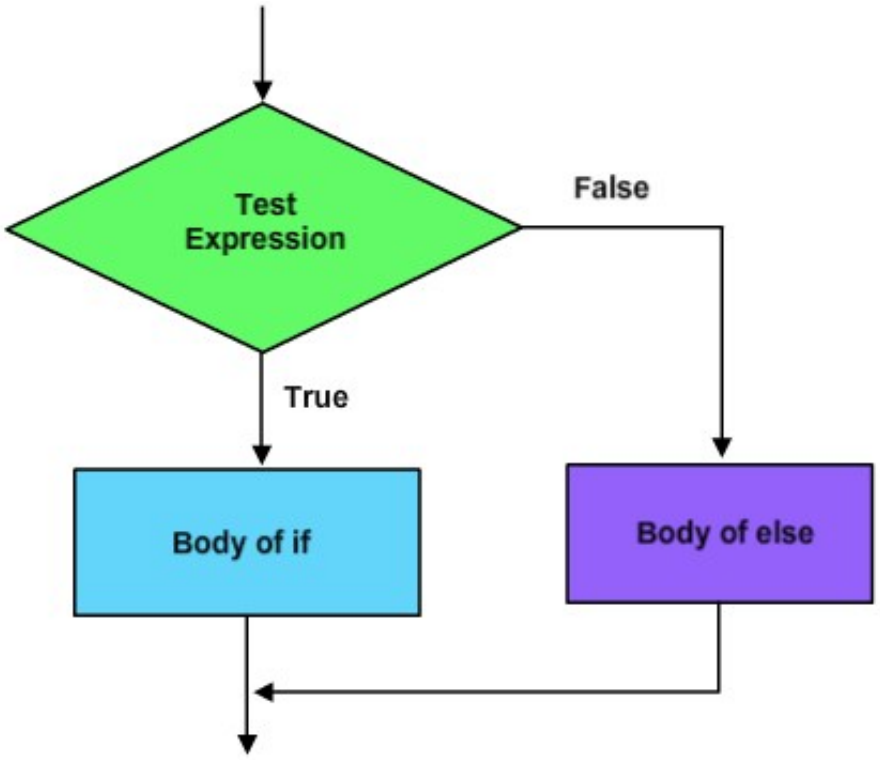
```
Variable x is less than y
```

Else Statement

- An ‘if’ statement can optionally include an else clause.
- The else clause is included as shown below:
  - If expression evaluates to true, statement1 is executed.
  - If expression evaluates to false, statement2 is executed.
  - Both statement1 and statement2 can be compound statements or blocks.

If-else Statement

- The combination of the ‘if’ and else clause is called the ‘if-else’ statement.



Operation of if ... else statement

- This is the most common form of the 'if' statement.
- If expression is true, statement1 is executed; otherwise, statement2 is executed.
- This is a nested if.
- If the first expression, expression1, is true, statement1 is executed before the program continues with the next statement.
- If the first expression is not true, the second expression, expression2, is checked.
- If the first expression is not true, and the second is true, statement2 is executed.
- If both expressions are false, statement3 is executed.
- Only one of the three statements is executed.

If-else statement Syntax

```
if (condition)
{
    /*Control will come inside only when the above condition is true*/
    //C statement(s)
}
else
{
    /*Control will come inside only when condition is false */
    //C statement(s)
}
```

If-else statement: Example

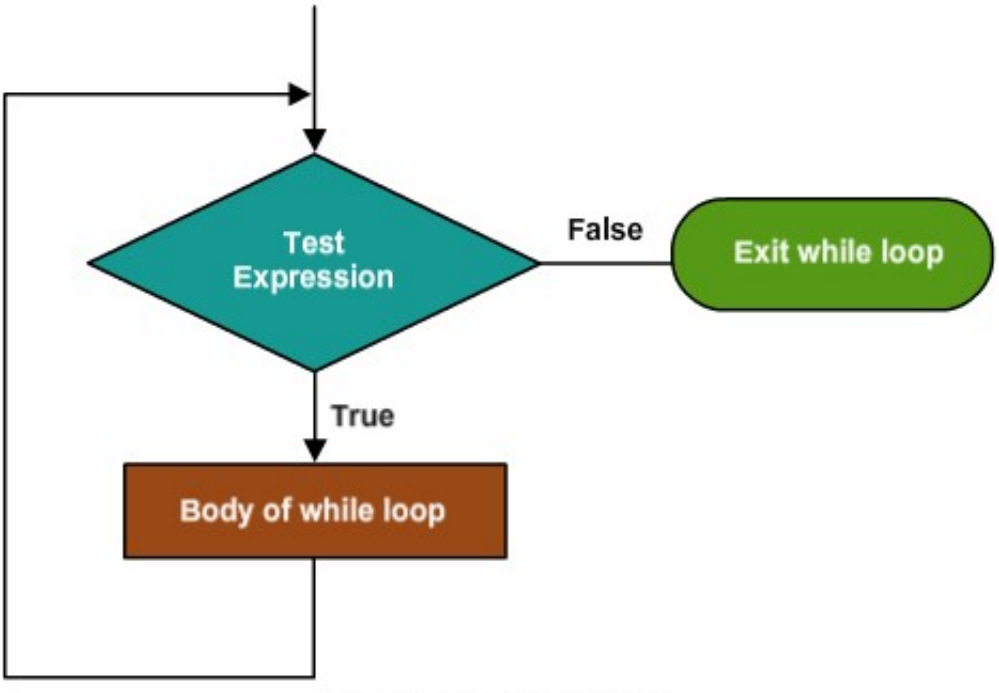
```
..
if(num >10)
   printf("num is greater than 10");
else
   printf("num is less than or equal to 10");
..
```

Loops

- There may be a situation, when you need to execute a block of code several numbers of times.
- In general, statements are executed sequentially.
- The first statement in a function is executed first, followed by the second, and so on.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages.
- C programming language provides the following types of loop to handle looping requirements.
  - while loop.
  - for loop.
  - do…while loop.

While loop - While Statement

- The 'while' statement, also called the while loop, executes a block of statements as long as a specified condition is true.
- The while statement has the following form:



Flowchart of while loop

- Condition is any C expression, and statement is a single or compound C statement.
- When program execution reaches a while statement, the following events occur:

- The expression condition is evaluated.
- If condition evaluates to false (that is, zero), the while statement terminates, and execution passes to the first statement following the while statement.
- If condition evaluates to true (that is, nonzero), the C statement(s) in statement are executed.
- Execution returns to step 1.

while loop: Syntax

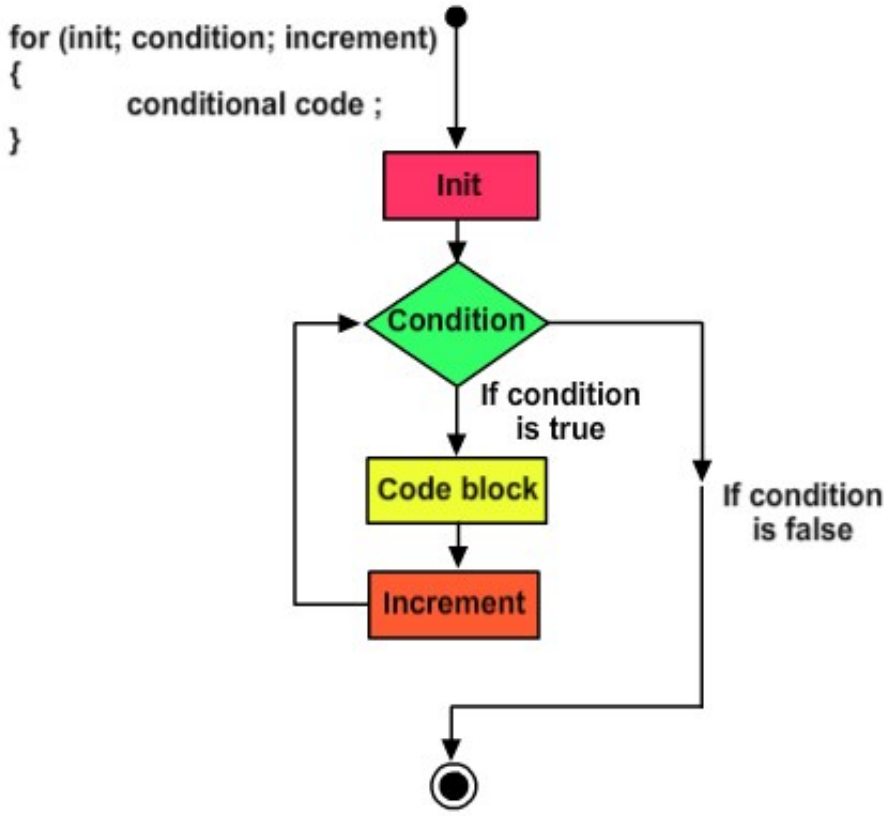
```
while (condition test)
{
    // C- statements, which requires repetition.
    // Increment (++) or Decrement (--) Operation.
}
```

while loop: Example

```
..
main()
{
   int count=1;
   while (count <=4)
   {
      printf("%d ", count);
      count++;
   }
}
```

For loop - Introduction

- The ‘for’ statement is a C programming construct that executes a block of one or more statements a certain number of times.
- It is sometimes called the ‘for’ loop because program execution typically loops through the statement more than once.
- A for statement has the following structure:



- Initial, condition, and increment are all C expressions, and statement is a single or compound C statement.

The 'for' Statement

- When a 'for' statement is encountered during program execution, the following events occur:
  - The expression initial is evaluated. Initial is usually an assignment statement that sets a variable to a particular value.
  - The expression condition is evaluated; condition is typically a relational expression.
  - If condition evaluates to false (that is, as zero), the 'for' statement terminates, and execution passes to the first statement following the for statement.
  - If condition evaluates to true (that is, as nonzero), the C statement(s) in statement are executed.
  - The expression increment is evaluated, and execution returns to the expression condition.

for loop: Syntax

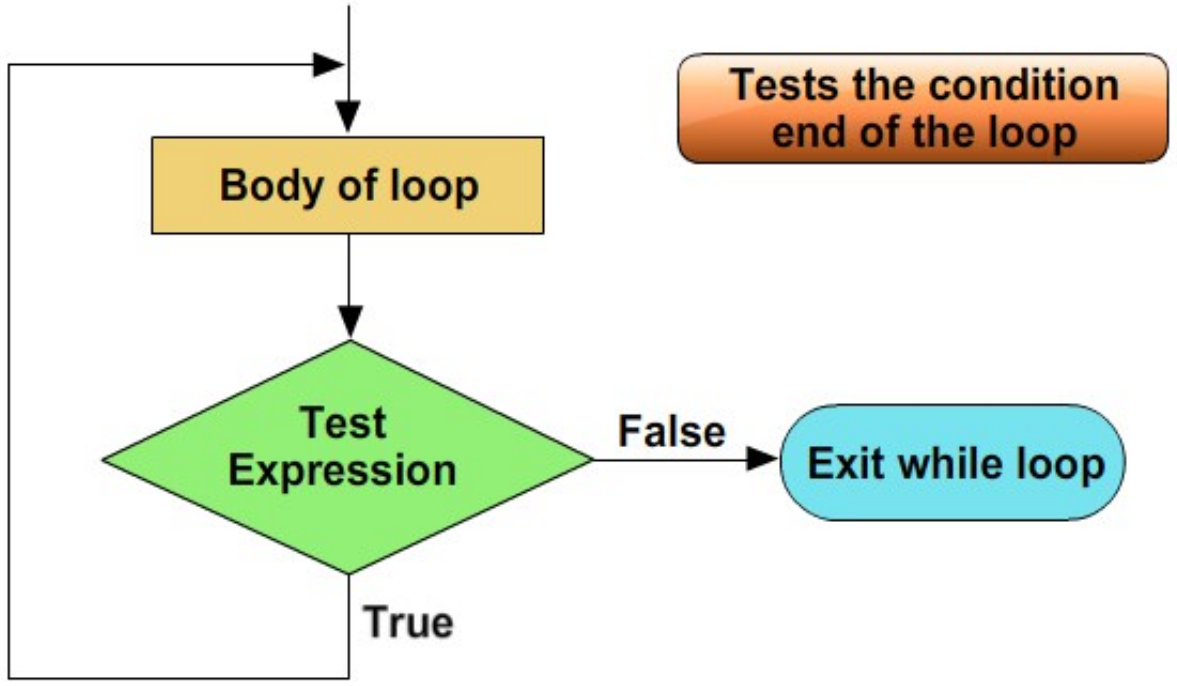
```
for(initialization; condition test; increment or decrement)
{
    //Code – C statements needs to be repeated
}
```

for loop: Example

```
...
int i;
for(i=1;i<=3; i++)
{
   printf("hello, World");
}
...
```

The do...while Loop

- C's third loop construct is the 'do...while'loop, which executes a block of statements as long as a specified condition is true.
- The do...while loop tests the condition at the end of the loop rather than at the beginning, as is done by the 'for' loop and the while loop.
- The structure of the do...while loop is as follows:



- Condition is any C expression, and statement is a single or compound C statement.
- When program execution reaches a do...while statement, the following events occur:
  - Step 1:The statements in statement are executed.
  - Step 2:Condition is evaluated.
  - Step 3:If it's true, execution returns to step 1.
  - Step 4:If it's false, the loop terminates.
- The statements associated with a do...while loop are always executed at least once.
- This is because the test condition is evaluated at the end, instead of beginning, of the loop.

  In contrast, for loops and while loops evaluate the test condition at the start of the loop, so the

- associated statements are not executed at all if the test condition is initially false.
- The do...while loop is used less frequently than while and for loops.
- It is most appropriate when the statement(s) associated with the loop must be executed at least once.
- The user could, of course, accomplish the same thing with a while loop by making sure that the test condition is true when execution first reaches the loop.

do...while loop: Syntax

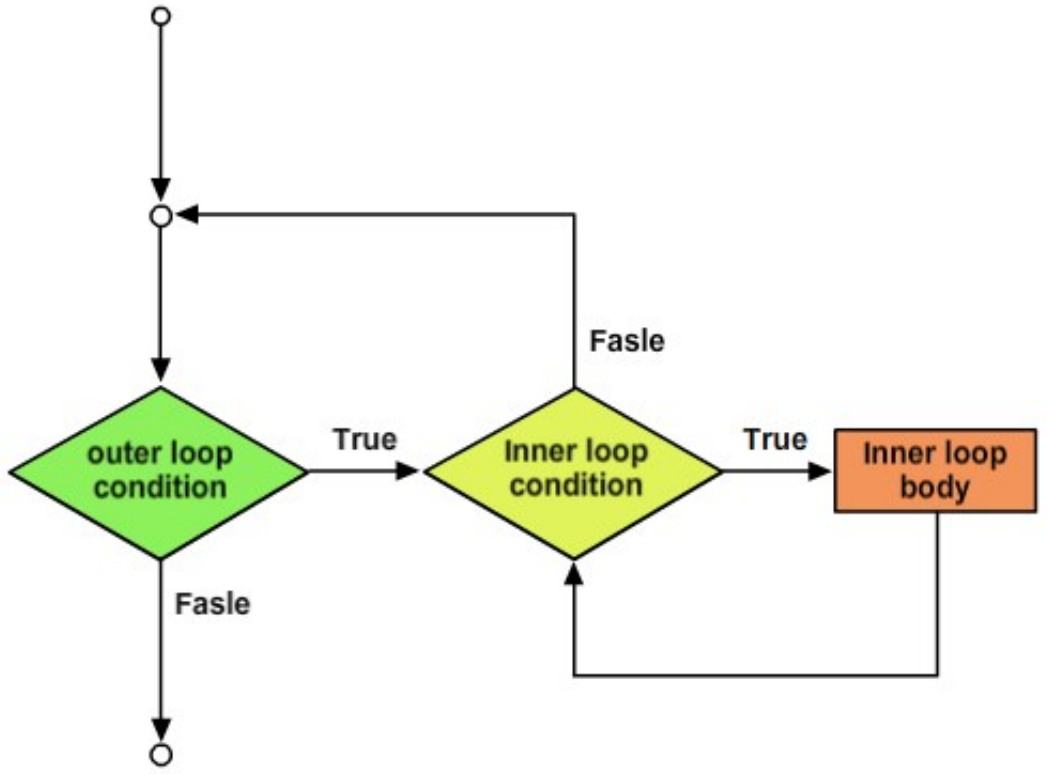
```
..
do
{
   //C- statements

}while(condition test);
```

do...while loop: Example

```
main()
{
    int i=0
    do
    {
      printf("while vs do-while\n");
    }while(i==1);
    printf("Out of loop");
}
```

Nested Loop - Introduction

- A nested loop is a loop within a loop, an inner loop within the body of an outer one.
- It has the following structure:



- What happens is that the first pass of the outer loop triggers the inner loop, which executes to completion.
- Then the second pass of the outer loop triggers the inner loop again.
- This repeats until the outer loop finishes.
- Of course, a break within either the inner or outer loop may interrupt this process.
- When the user "nest" two loops, the outer loop takes control of the number of complete repetitions of the inner loop.
- While all types of loops may be nested, the most commonly nested loops are for loops.

Example

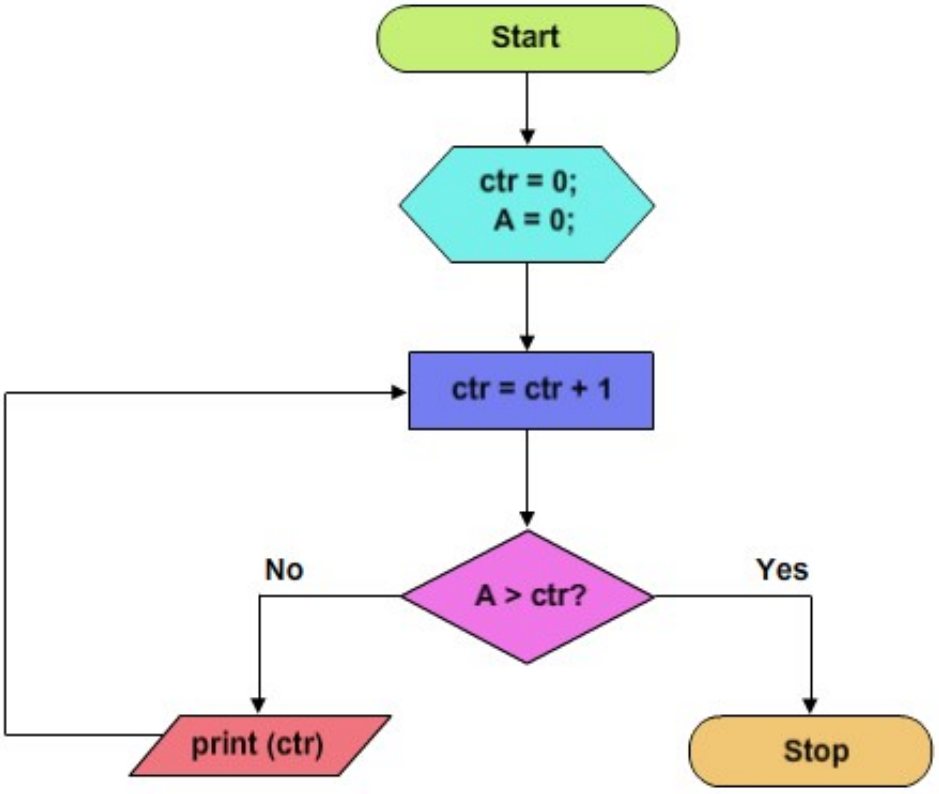
```
main()
{
    for (int i=0; i<=10; i++)
    {
        for (int j=0; j<=10; j++)
        {
            printf("%d, %d",i ,j);
        }
    }
}
```

Infinite Loop

- An ‘infinite’loopis a loop that never terminates.
- When the condition that controls the loop execution never becomes false, the loop can never

stop executing.

- One such example is shown below:



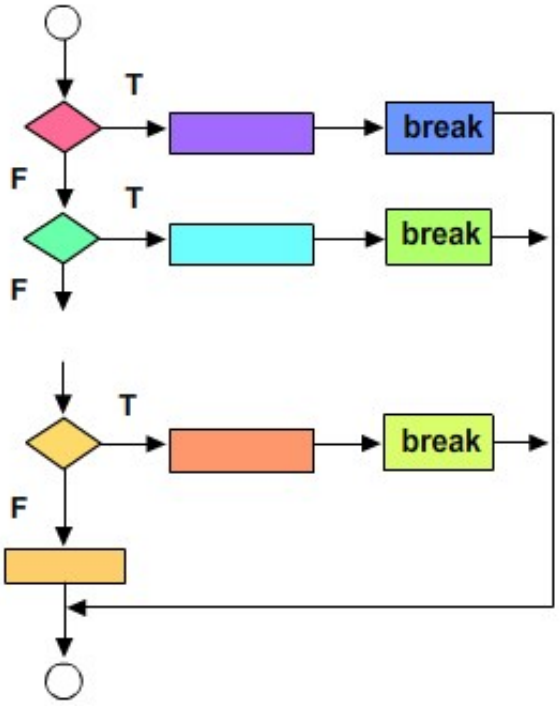
- One creates an infinite loop.
- The condition that the while tests is the constant 1, which is always true and can't be changed by the program.
- Because 1 can never be changed on its own, the loop never terminates.
- One can also create an infinite loop with the for-statement:

Switch Statement

- C's most flexible program control statement is the switch statement, which lets the program execute different statements based on an expression that can have more than two values.
- Earlier control statements, such as if, were limited to evaluating an expression that could have only two values:
  - True or false.
- To control program flow based on more than two values, the user had to use multiple nested if statements.
- Its purpose is to allow the value of a variable or expression to control the flow of program execution.
- A switch statement is defined across many individual statements.

Switch Case

- The structure of switch case is as shown below:



```
switch(expression)
{
case constant1:
statements 1;
break;
case constant2:
statements 2;
break;
.....
default:
statements n;
break;
}
```

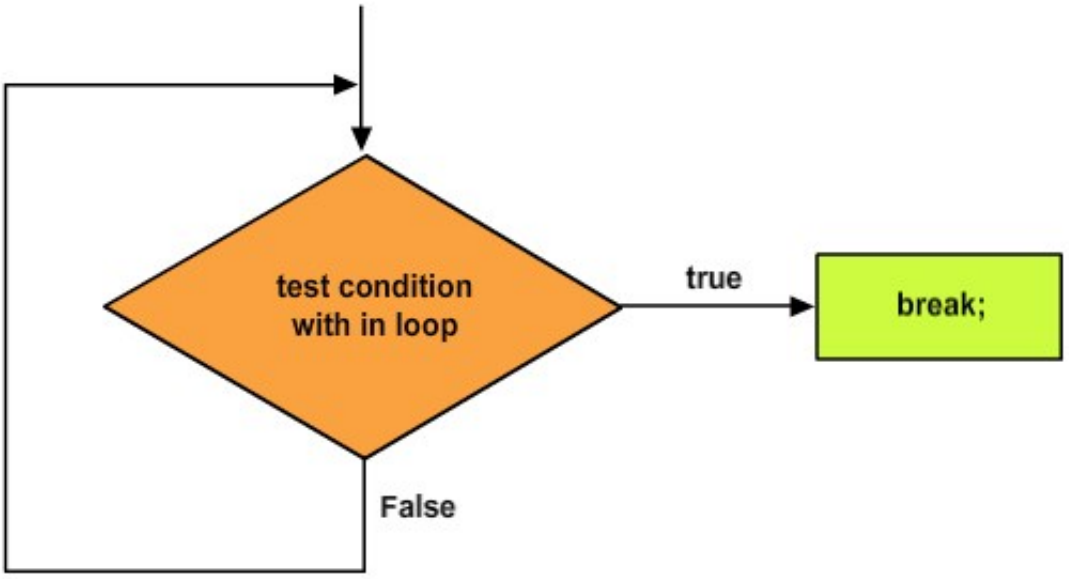
- From the statement, expression is any expression that evaluates to an integer value: type long, int, or char.
- The switch statement evaluates expression and compares the value against the templates following each case label, and then one of the following happens:
- If a match is found between expression and one of the templates, execution is transferred to the statement that follows the case label.
- If no match is found, execution is transferred to the statement following the optional default label.
- If no match is found and there is no default label, execution passes to the first statement following the switch statement's closing brace.

switch case: Example

```
int main()
{
    int num=2;
    switch(num+2)
    {
       case 1:
        printf("Case1: Value is: %d", num);
       case 2:
        printf("Case1: Value is: %d", num);
       case 3:
        printf("Case1: Value is: %d", num);
       default:
        printf("Default: Value is: %d", num);
    }
    return 0;
}
```

Break Statement

- The break statement can be placed only in the body of a ‘for’ loop, ‘while’ loop, or ‘do...while’ loop.

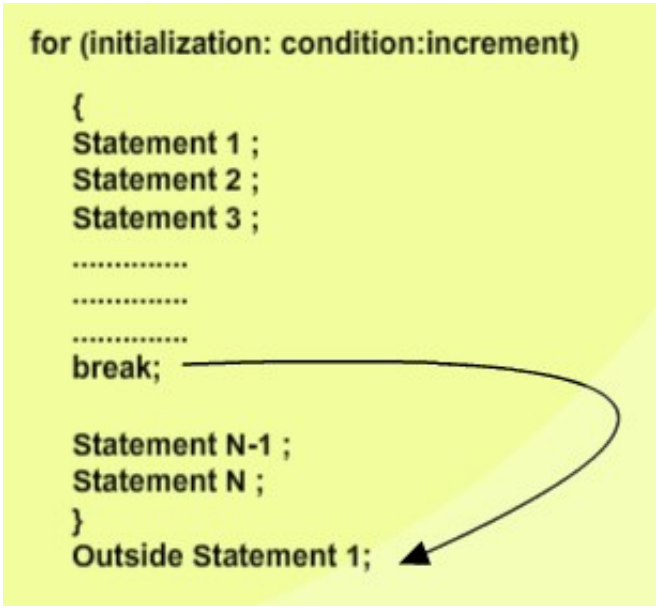


Break statement

- When a break statement is encountered, execution exits the loop.

```
int main()
{
int i;
clrscr();
for(i=1;i<=10;i=i+1)
{
if(i==2)
break;
printf("%d\n",i);
}
while(i<=10)
{
if(i==4)
break;
printf("%d\n",i);
i=i+1;
}
d0
{
if(i==6)
break;
printf("%d\n",i);
i=i+1;
}
while(i<=10)
getch();
return 0;
}
```

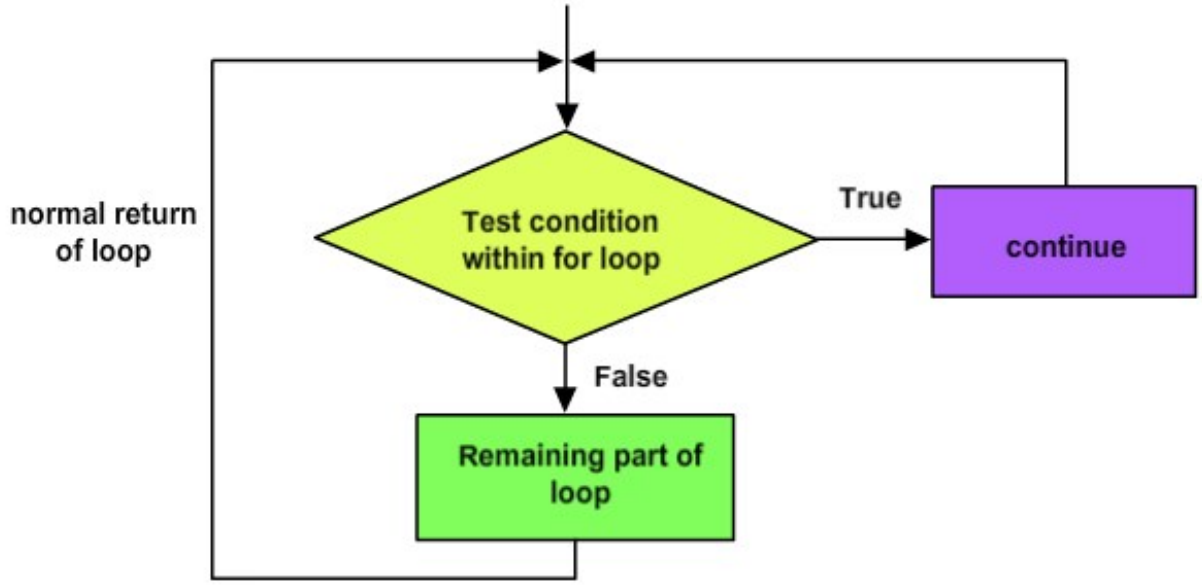
- From the example, left to itself, the 'for' loop would execute 10 times.
- On the sixth iteration, however, count is equal to 5, and the break statement executes, causing the 'for' loop to terminate.
- Execution then passes to the statement immediately following the for loop′s closing brace.
- When a break statement is encountered inside a nested loop, it causes the program to exit the innermost loop only.



- The break statement ends the loop and moves control to the next statement outside the loop.
- The break statement ends only the smallest enclosing do, for, switch, or while statement.
- The break statement in C programming language has the following two usages:
  - When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
  - It can be used to terminate a case in the switch statement.

Continue Statement

- Like the break statement, the continue statement can be placed only in the body of a ‘for’ loop, a while loop, or a ‘do...while’ loop.



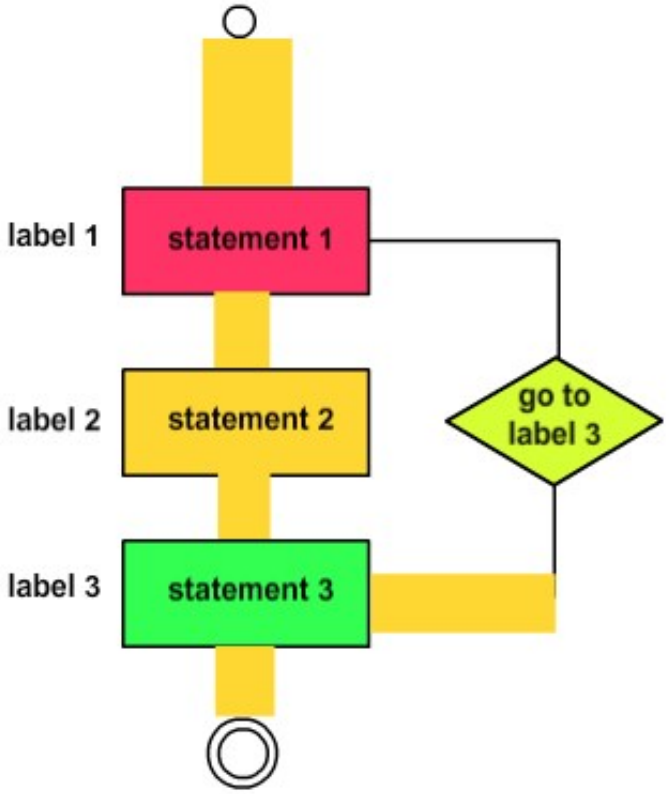
Flowchart of continue statement

- When a continue statement executes, the next iteration of the enclosing loop begins immediately.
- The statements between the continue statement and the end of the loop aren't executed
  - Continue;
- Continue is used inside a loop.
- It causes the control of a program to skip the rest of the current iteration of a loop and start the next iteration.
- The following is an example for the continue statement.
  - Example:

```
..
for (int j=0; j<=8; j++)
{
   if (j==4)
   {
       continue;
   }

   printf("%d ",j);
}
..
```

Goto Statement

- The goto statement is one of C's unconditional jump, or branching statements.



- When program execution reaches a goto statement, execution immediately jumps, or branches, to the location specified by the goto statement.
- This statement is unconditional because execution always branches when a goto statement is encountered; the branch doesn't depend on any program conditions.
- A goto statement and its target must be in the same function, but they can be in different blocks.
- Often, a break statement, a continue statement, or a function call can eliminate the need for a goto statement.

Goto Statement: Example

```
int main()
{
   int age;
   Vote:
     printf("you are eligible for voting");

   NoVote:
     printf("you are not eligible to vote");

   printf("Enter you age:");
   scanf("%d", &age);
   if(age>=18)
        goto Vote;
   else
        goto NoVote;

   return 0;
}
```

Structured Programming

- Structured programming is a methodology that is part of a renewed emphasis on software engineering, which involves the systematic design and development of software and the management of the software development process.
- Software engineering views the development of a program as a coordinated activity involving people, tools, and practices; use of modern design; and development and management methods in an integrated approach.
- Structured approach involves the use of methods such as top-down program design and a limited number of control structures in a program to create tightly structured modules of program code.
- The effect of top-down design and structured programming has been used to lower the overall cost of programming.
- The structured approach promises to reduce the cost of developing and maintaining computer programs by standardizing program development and structures used.
- This increases the simplicity and accuracy, and at the same time minimizes programming and maintenance cost.
- A 'traditional' flexible and creative environment provided to the programmer often results in complex and difficult-to-read programs requiring much testing before they are error-free.
- These also become costly to develop and maintain.
- Structured programming, on the other hand, emphasizes group responsibility for program development.
- It also brings in a standardization of program-design concepts and methods, which significantly reduces the program complexity.
- Organizations using structured programming have shown the following characteristics:
  - Programming Productivity: Programmers write more program statements per day with fewer errors.
  - Program Economy: The cost and time of program development and maintenance are reduced.
  - Program Simplicity: Programs are easier to read, write, correct, and maintain.
  - These results highlight the reasons for structured programming to continue to be a popular programming methodology.

Modular Design

- Structured programming is used as a set of tools to improve program organization, facilitate problem solving, and make code easier to write and analyses in both individual and group

projects.

- Using structured programming, the solution to the problem is divided into segments called modules.
- A module is a logically separable part of a program.
- It is a unit, discrete and identifiable with respect to compilation and loading.
- In terms of common programming language constructs, a module can be a macro, a function, a procedure (or subroutine), a process, or a package.
  - Each module involves processing of data that are logically related. Modules are functional parts that aid in processing.
  - Ideally, each module works independent of other modules, although this is sometimes impossible.
  - Modules are ranked by hierarchy and organized on the basis of importance.
  - The lower the module on the structure organization plan, more is the detail given to the programming steps involved.
  - The controlling module resides at the top level.
- It gives the view of the overall structure for the entire program.
- The system is designed to give more detail at each module level.
- A module is coded and tested, and then tested with other tested modules.
- This procedure makes program testing easier, since there is only one entry point and one exit point per module.
- C is called a structured programming language because to solve a large problem, C programming language divides the individual small responsibilities to smaller modules called functions or procedures.
- One major drawback of C language is that similar functions cannot be grouped inside a module or class.
- Also functions cannot be associated to a type or structure thus data and functions cannot be bound together.

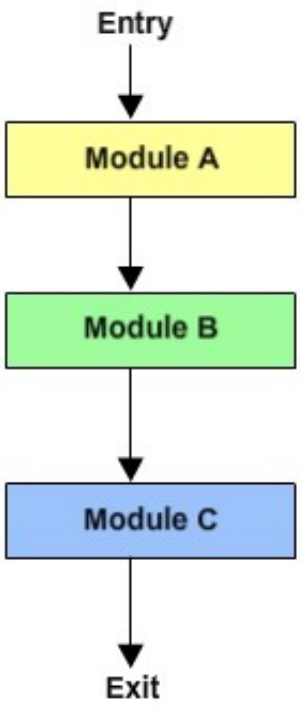
Structured Programming Constructs

- The most common techniques used in structured programming to solve all problems are called constructs.
- Sequence, selection, and repetition
- These constructs are also called control structures.

- Using these three basic control structures, it is possible to write standardized programs, which are easy to read and understand.

Sequence Structure

- Sequence refers to an instruction or a series of instructions that perform a required calculation or allow input or output of data.
- Since these steps are executed one after the other, there is no change in the flow logic.



- The picture illustrates that program statements in function A will be executed before those for function B.
- In other words, we say that control 'flows' from function A to function B.

Selection Structure

- Selection refers to testing for a certain condition for data.
- There are only two possible answers to questions regarding data – true (yes) or false (no).
- One selection-technique variation is known as the IF-THEN-ELSE.
- The instructions that are to be executed when the condition is true follow the IF-THEN alternative.
- The instructions followed by the ELSE alternative represent what is to be executed when the condition is false.
- If the condition is true, the control will flow to function B and its statements will be executed; if it is false, function A is executed.

Repetition Structure

- Repetition involves the use of a series of instructions that are repeated until a certain condition is met.
- Repetition involves the use of two variations – the for, while and the do-while.
- The while performs a function as long as a condition is true.
- On the other hand, do-while allows a function to be executed until the given condition is false.
- Another marked difference is that the while first tests the given condition and then executes the function, whereas do-while processes the function before checking the condition.
- Before writing a program to solve a particular problem, it's essential to have a thorough understanding of the problem and a carefully planned approach to solving the problem.

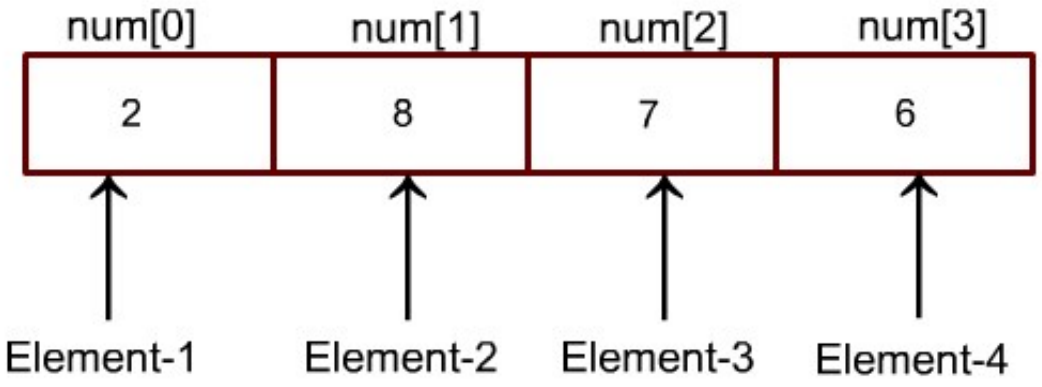
Control Structures

- Normally, statements in a program are executed one after the other in the order in which they're written.
- This is called sequential execution.
- Various C statements we'll soon discuss enable the user to specify that the next statement to be executed may be other than the next one in sequence.
- This is called transfer of control.

UNIT - 5
Arrays

Array

- An array is a collection of same type of elements which are sheltered under a common name.
- An array can be visualized as a row in a table, whose each successive block can be thought of as memory bytes containing one element.
- Look at the figure below :
  - An Array of four elements:



- The number of 8 bit bytes that each element occupies depends on the type of array.
- If type of array is ‘char’ then it means the array stores character elements.
- Since each character occupies one byte so elements of a character array occupy one byte each.
- An array is a collection of data storage locations, each having the same data type and the same name.
- Each storage location in an array is called an array element.
- Arrays are of two types:
  - Single or One-dimensional arrays.
  - Multidimensional eg.two dimensional arrays.

Single-Dimensional Arrays

- A single-dimensional array has only a single subscript.
- A subscript is a number in brackets that follows an array name.
- This number can identify the number of individual elements in the array.
- An example should make this clear.
- For the company expenditure program, the users could use the following line to declare an array of type float:

- float expense[12];

- The array is named expense, and it contains 12 elements.
- Each of the 12 elements is the exact equivalent of a single float variable.
- All of C′s data types can be used for creating arrays.
- C array elements are always numbered starting at 0, so the 12 elements of expense are numbered 0 through 11.
- In the above example, January′s expenditure would be stored in expense [0], February′s in expense [1], and so on.
- When an array is declared, the compiler sets aside a block of memory large enough to hold the entire array.
- Individual array elements are stored in sequential memory locations.

Naming and Declaring Arrays

- The rules for assigning names to arrays are the same as for variable names.
- An array name must be unique in that program which means should not be used for another array or for any other identifier (variable, constant).
- The array declaration consists of an array name followed by the number of elements in the array must be enclosed in square brackets.
- When the users declare an array, he can specify the number of elements with a literal constant or with a symbolic constant created with the #define directive.

Array elements

- Size of array defines the number of elements in an array.
- Each element of array can be accessed and used by user according to the need of program.
- For example:

- int age[5];

Array Elements

- Note that, the first element is numbered 0 and so on.
- Here, the size of array age is 5 times the size of int because there are 5 elements.
- Suppose, the starting array address of age[0] is 2120d and the size of int be 4 bytes.

- Then, the next address (address of a[1]) will be 2124d, address of a[2] will be 2128d and so on.

Accessing array elements

- An element is accessed by indexing the array name.
- This is done by placing the index of the element within square brackets after the name of the array.
- For example:
  - double salary = balance[9];
- The above statement will take 10th element from the array and assign the value to salary variable.
- Following is an example which will use three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>
int main ()
{
int n[ 10 ]; /* n is an array of 10 integers */
int i,j;
   /* initialize elements of array n to 0 */
   for(i = 0; i< 10; i++)
   {
     n[ i ] = i + 100; /* set element at location i to i + 100 */
   }
   /* output each array element's value */
   for(j = 0; j < 10; j++)
   {
printf("Element[%d] = %d\n", j, n[j]);
   }
   return 0;
}
```

- When the above code is compiled and executed, it produces the following result:
  - Element[0] = 100.
  - Element[1] = 101.
  - Element[2] = 102.
  - Element[3] = 103.
  - Element[4] = 104.
  - Element[5] = 105.
  - Element[6] = 106.

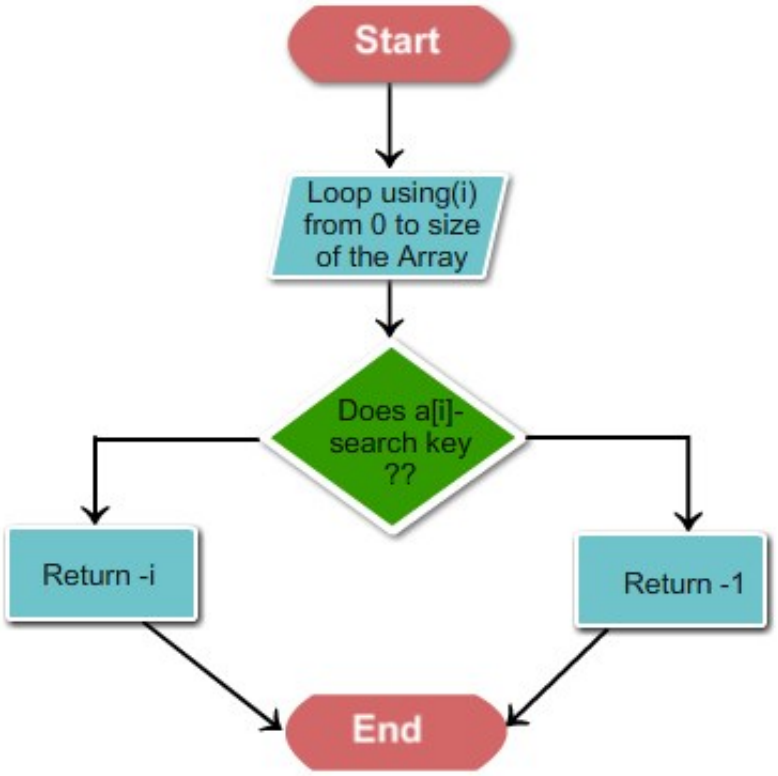
- Element[7] = 107.
- Element[8] = 108.
- Element[9] = 109.

Searching an Array - Introduction

- A common array operation is to search the values of an array for a particular value.
- Users may be doing this in order to find and change a value or to find and process information in a corresponding element of a paired or parallel array.

A linear Search of an Array

- A linear search starts searching for the value with the first element of the array and continues through the array, one element at a time until the wanted value is found or the user reach the end of the array.
- Linear search is also known as sequential search that is suitable for searching a list of data for a particular value.
- Listed below is the flow for a linear search in an array.



- Example:

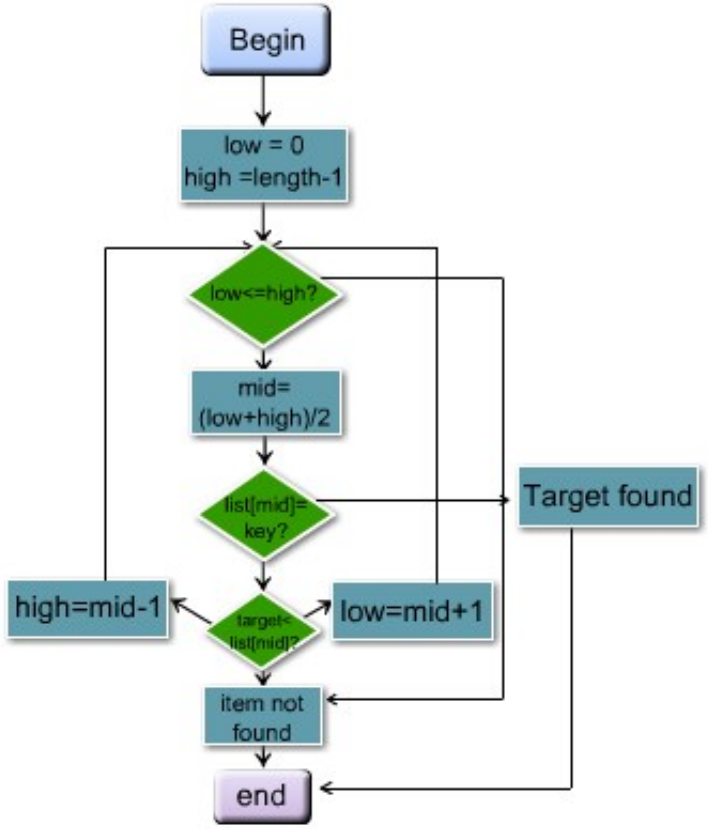
```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int arr[50],temp[50],i,j,item,no,t;
char ch;
cout<<"\n ENTER HOW MANY ELMENT TO BE PROCESSED:?\n";
cin>>no;
cout<<"\n ENTER YOUR ELEMENT\n";
for(i=1;i<=no;i++)
 {
 cin>>arr[i];
 }
cout<<"\n THE ARRAY ELEMENTS ARE:...\n";
for(j=1;j<=no;j++)
 {
 cout<<"\t"<<arr[j]<<endl;
 }
int a=1;
do
{
cout<<"\n\n\n\n\n ENTER YOUR ELMENT TO BE SEARCHED ?\n";
cin>>item;
do
{
 while(arr[a]!=item)
 {
 a=a+1;
 }
if(a>no)
 {
 cout<<"\n Search is unsucces full...\n";
 }
 else
 {
 int loc=a;
 cout<<"\n Item is found in loation...: "<<loc<<endl;
 }
}
while(i<no);
cout<<"\n DO YOU FINISHED...?\n OR YOU WANT TO USE PROGRAM again?
(press Y / N)\n";
cin>>ch;
}
while(ch=='y');
}
```

A Binary Search of an Array

- A binary search requires that the element values in the array be in order, such as ascending order, from smallest to largest.
- A binary search algorithm (or binary chop) is a technique for locating a particular value in a

sorted list.

- The method makes progressively better guesses, and closes in on the location of the sought value by selecting the middle element in the span (which, because the list is in sorted order, is the median value), comparing its value to the target value, and determining if it is greater than, less than, or equal to the target value.
- A guessed index whose value turns out to be too high becomes the new upper bound of the span, and if its value is too low that index becomes the new lower bound.
- Now, the user can search the array by:
  - Finding the center element value.
  - If this is the wanted value, the users are done.
  - Otherwise, if they required value is greater than this value and it lies in the top half of the array it eliminates the bottom half of the array and it finds the new center of what is left.
  - Otherwise, if they wanted value is less than this value and it lies in the bottom half of the array, it eliminates the top of the array and finds the new center of what is left.
  - This is done until the user find the value or he run out of array.
- Listed below is the flow for a binary search in an array:

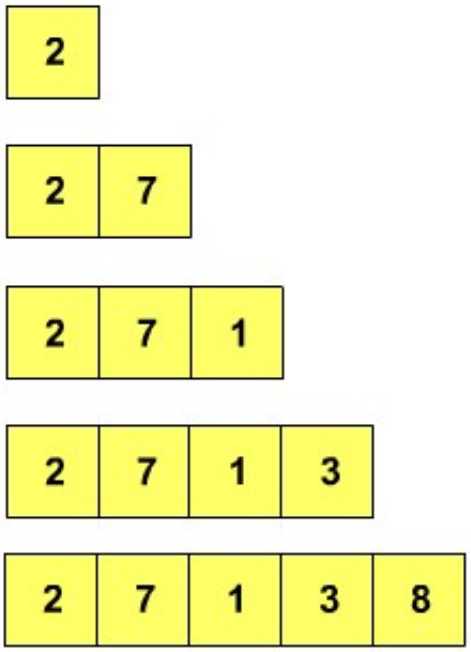


Example: Binary Search

```
include<stdio.h>
int main(){
    int a[10],i,n,m,c=0,l,u,mid;
    printf("Enter the size of an array: ");
    scanf("%d",&n);
    printf("Enter the elements in ascending order: ");
    for(i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    printf("Enter the number to be search: ");
    scanf("%d",&m);
    l=0,u=n-1;
    while(l<=u){
        mid=(l+u)/2;
        if(m==a[mid]){
            c=1;
            break;
        }
        else if(m<a[mid]){
            u=mid-1;
        }
        else
            l=mid+1;
    }
    if(c==0)
        printf("The number is not found.");
    else
        printf("The number is found.");
    return 0;
}
```

Inserting Elements into an Array

- Given below is the flow for inserting the values into an array at run time.



- Example:

```
#include <stdio.h>
 void main()
{
   int array[10];
   int i, j, n, m, temp, key, pos;
    printf("Enter how many elements \n");
   scanf("%d", &n);
   printf("Enter the elements \n");
   for (i = 0; i < n; i++)
   {scanf("%d", &array[i]);}
   printf("Input array elements are \n");
   for (i = 0; i < n; i++)
   {printf("%d\n", array[i]);}
   for (i = 0; i < n; i++)
   { for (j = i + 1; j < n; j++)
      {  if (array[i] > array[j])
         {  temp = array[i];
            array[i] = array[j];
            array[j] = temp;
         }    }   }
   printf("Sorted list is \n");
   for (i = 0; i < n; i++)
   { printf("%d\n", array[i]); }
   printf("Enter the element to be inserted \n");
   scanf("%d", &key);
   for (i = 0; i < n; i++)
   { if (key < array[i])
      {  pos = i;
         break;    }  }
   m = n - pos + 1 ;
   for (i = 0; i <= m; i++)
   { array[n - i + 2] = array[n - i + 1] ; }
   array[pos] = key;
   printf("Final list is \n");
   for (i = 0; i < n + 1; i++)
   { printf("%d\n", array[i]); }
}
```

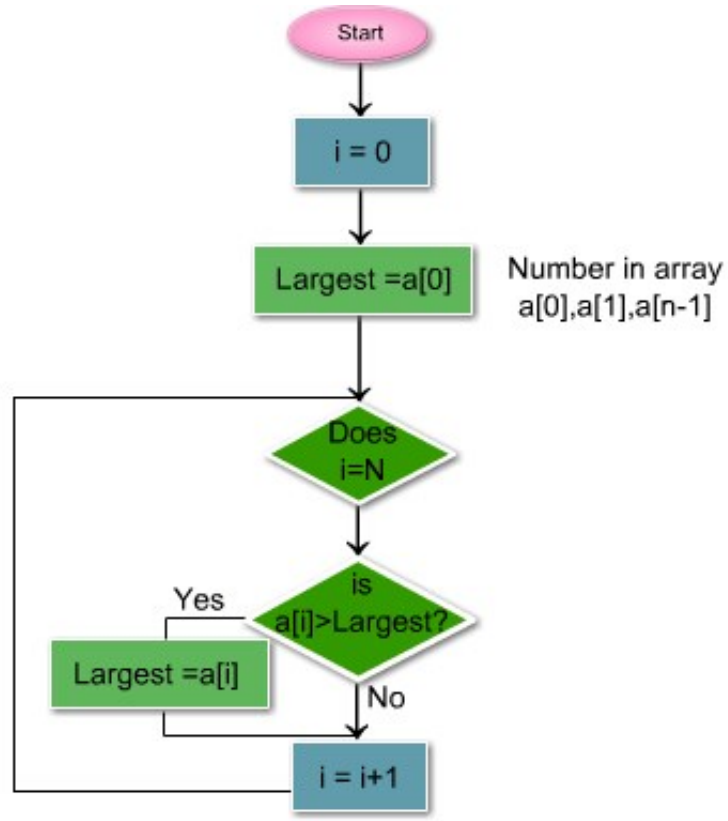
Deleting Elements from an Array

- Example:

```
#include <stdio.h>

int main()
{
  int array[100], position, c, n;

  printf("Enter number of elements in array\n");
  scanf("%d", &n);

  printf("Enter %d elements\n", n);

  for ( c = 0 ; c < n ; c++ )
    scanf("%d", &array[c]);

  printf("Enter the location where you wish to delete element\n");
  scanf("%d", &position);

  if ( position >= n+1 )
    printf("Deletion not possible.\n");
  else
  {
    for ( c = position - 1 ; c < n - 1 ; c++ )
      array[c] = array[c+1];

    printf("Resultant array is\n");

    for( c = 0 ; c < n - 1 ; c++ )
      printf("%d\n", array[c]);
  }

  return 0;
}
```

Find the Largest/Smallest of the Elements of an Array

- This time, we are looking for the largest value in the array.
- We just set largest to the value of the first element of the array.
- Then we compare this value to each of the other elements in the array.
- If one is larger, we replace the value in largest with the value and continue to check the rest of the array.
- The following the flow to find the largest of the elements of an array.

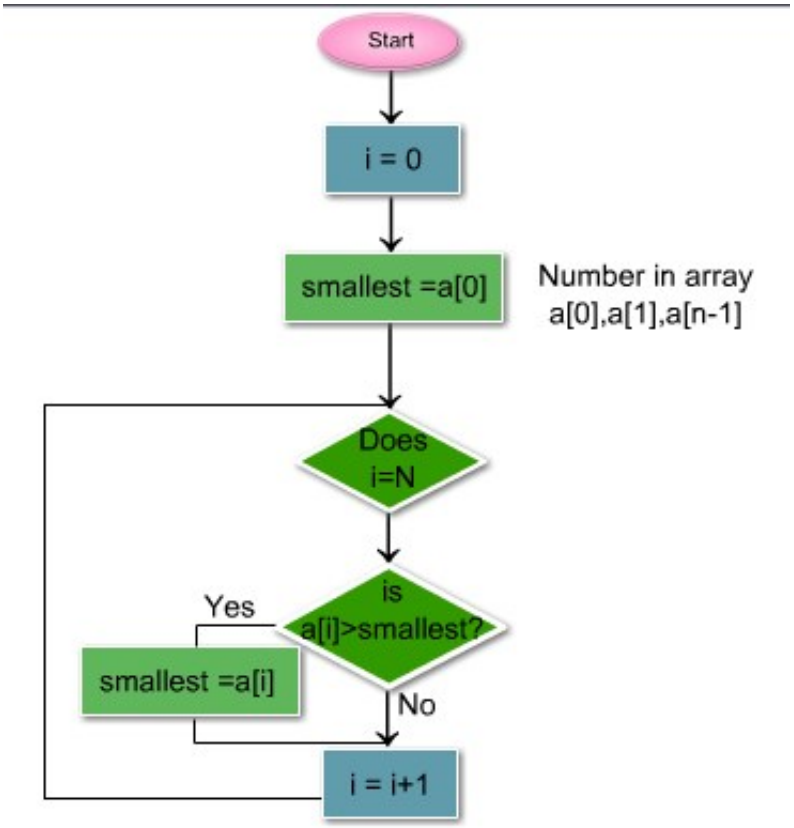


Example:

```
#include <stdio.h>

int main()
{
 int array[100], maximum, size, c, location = 1;

 printf("Enter the number of elements in array\n");
 scanf("%d", &size);

 printf("Enter %d integers\n", size);

 for (c = 0; c < size; c++)
  scanf("%d", &array[c]);

 maximum = array[0];

 for (c = 1; c < size; c++)
 {
  if (array[c] > maximum)
  {
    maximum  = array[c];
    location = c+1;
  }
 }

 printf("Maximum element is present at location %d
 and it's value is %d.\n", location, maximum);
 return 0;
}
```

Find the Smallest of the Elements of an Array

- We set smallest to the value of the first element of the array and start comparing it to other elements in the array.
- The following is the flow to find the smallest of the elements of an array:



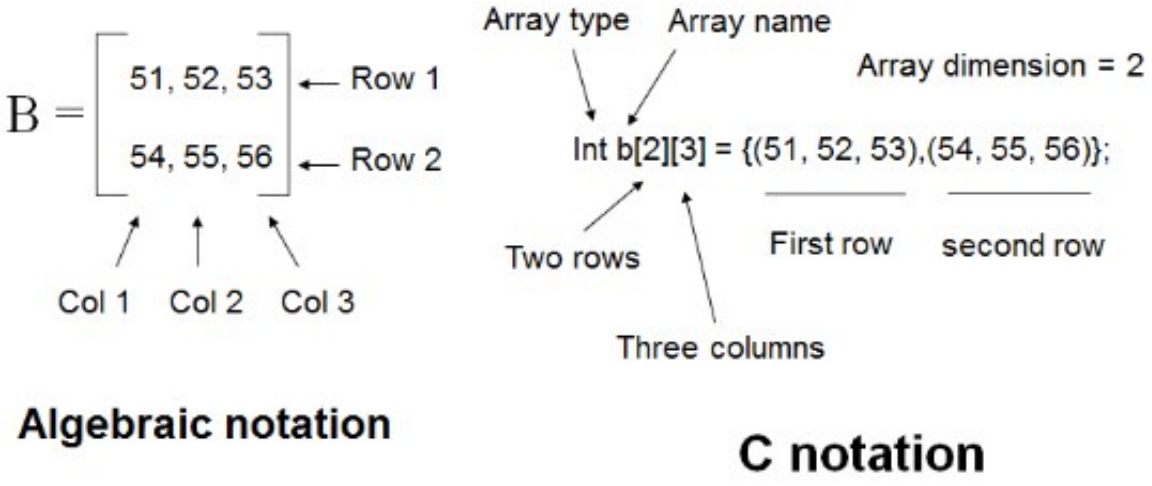
Example:

```
#include <stdio.h>

int main()
{
   int array[100], minimum, size, c, location = 1;

   printf("Enter the number of elements in array\n");
   scanf("%d",&size);

   printf("Enter %d integers\n", size);

   for ( c = 0 ; c < size ; c++ )
      scanf("%d", &array[c]);

   minimum = array[0];

   for ( c = 1 ; c < size ; c++ )
   {
      if ( array[c] < minimum )
      {
         minimum = array[c];
         location = c+1;
      }
   }

   printf("Minimum element is present at location %d and it's value is %d.\n",
    location, minimum);
   return 0;
}
```

Introduction to Multidimensional Arrays - Two-Dimensional Arrays

- A two-dimensional array has two subscripts.
- A two-dimensional array has a row-and-column structure as shown below:
  - int matrix[row][column];
- Where row can only be an integer.
- And column can only be an integer.

What is a Two-dimensional array?



Indexes in 2D arrays

- Assume that the two dimensional array called val is declared and looks like the following:

val	Col 0	Col 1	Col 2	Col 3
Row 0	8	16	9	52
Row 1	3	15	27	6
Row 2	14	25	2	10

- To access the cell containing 6, we reference val[1][3],that is, row 1, column 3.

DECLARATION

- How to declare a multidimensional array?
  - int b[2][3];
    - The name of the array to be b.
    - The type of the array elements to be int.

- The dimension to be 2 (two pairs of brackets []).
- The number of elements or size to be 2*3 = 6.

INITIALIZATION

- How to initialize a Two-Dimensional array?
  - Initialized directly in the declaration statement.

```
int b[2][3] = {51,52,53,54,56};

b[0][0] = 51   b[0][1] = 52   b[0][2] = 53
```

  - Use braces to separate rows in 2-D arrays.

```
int c[4][3] =    { {1,2,3},
                   {4,5,6},
                   {7,8,9},
                   {10,11,12}};
int c[ ][3] =    { {1,2,3},
                   {4,5,6},
                   {7,8,9},
                   {10,11,12}};
```

  - Implicitly declares the number of rows to be 4.

Input of Two-Dimensional Arrays

- Data may be input into two-dimensional arrays using nested for loops interactively or with data files.
- A nested for loop is used to input elements in a two dimensional array.
- In this way by increasing the index value of the array the elements can be entered in a 2d array.

Output of Two-Dimensional Arrays

- The output of two-dimensional arrays should be in the form of rows and columns for readability.
- Nested for loops are used to print the rows and columns in row and column order.

- By increasing the index value of the array the elements stored at that index value are printed on the output screen.

Example: A program to input elements in a two dimensional array and print it

```
#include<stdio.h>
int main()
{
int i,j;
// declaring and Initializing array
int arr[2][2] = {10,20,30,40};
/* Above array can be initialized as below also
arr[0][0] = 10; // Initializing array
arr[0][1] = 20;
arr[1][0] = 30;
arr[1][1] = 40; */
for (i=0;i<2;i++)
{
for (j=0;j<2;j++)
{
// Accessing variables
printf("value of arr[%d] [%d] : %d\n",i,j,arr[i][j]);
}
}
}
```

Output:

```
value of arr[0] [0] is 10
value of arr[0] [1] is 20
value of arr[1] [0] is 30
value of arr[1] [1] is 40
```

Passing Arrays to Functions

- Two-dimensional arrays may by passed by array name.
- Because arrays are stored by rows, in order to accurately locate an element, a function must know the length of a row: that is the number of columns.
- This must be included in both the function prototype and the header of the function definition.
- Passing Fixed Sized Arrays.
- Passing Array Elements.

Multidimensional Array

- A multidimensional array has more than one subscript.
- A two-dimensional array has two subscripts.
- A three-dimensional array has three subscripts, and so on.
- There is no limit to the number of dimensions a C array can have.
- Example

```
#include<stdio.h>int main(){
int i;
int arr[5] = {10,20,30,40,50};
// declaring and Initializing array in C
//To initialize all array elements to 0, use int arr[5]={0};
/* Above array can be initialized as below also
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50; */
for (i=0;i<5;i++)
{
// Accessing each variable
printf("value of arr[%d] is %d \n", i, arr[i]);
}
}
```

Output:

```
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
```

Addition/Multiplication of Two Matrices

Addition of Two Matrices: Program and Output

```
#include <stdio.h>
 int main()
{
   int m, n, c, d, first[10][10], second[10][10], sum[10][10];
   printf("Enter the number of rows and columns of matrix\n");
   scanf("%d%d", &m, &n);
   printf("Enter the elements of first matrix\n");
    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
         scanf("%d", &first[c][d]);
    printf("Enter the elements of second matrix\n");
    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
           scanf("%d", &second[c][d]);
    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
         sum[c][d] = first[c][d] + second[c][d];
    printf("Sum of entered matrices:-\n");
 for ( c = 0 ; c < m ; c++ )
   {
     for ( d = 0 ; d < n ; d++ )
        printf("%d\t", sum[c][d]);

     printf("\n");
   }
    return 0;
}
```

Output:

```
Enter the number of rows and columns of matrix
2
2
Enter the elements of first matix
2
4
Enter the elements of second matrix
5   6
2   1
Sum of entered matices:-
6    8
5    5
```

Multiplication of Two Matrices: Program and Output

```
#include <stdio.h>
int main()
{
  int m, n, p, q, c, d, k, sum = 0;
  int first[10][10], second[10][10], multiply[10][10];
  printf("Enter the number of rows and columns of first matrix\n");
  scanf("%d%d", &m, &n);
  printf("Enter the elements of first matrix\n");
  for ( c = 0 ; c < m ; c++ )
    for ( d = 0 ; d < n ; d++ )
      scanf("%d", &first[c][d]);
  printf("Enter the number of rows and columns of second matrix\n");
  scanf("%d%d", &p, &q);
  if ( n != p )
    printf("Matrices with entered orders can't be multiplied with each other.\n");
  else
  {
    printf("Enter the elements of second matrix\n");
     for ( c = 0 ; c < p ; c++ )
      for ( d = 0 ; d < q ; d++ )
        scanf("%d", &second[c][d]);
     for ( c = 0 ; c < m ; c++ )
     {
      for ( d = 0 ; d < q ; d++ )
      {
        for ( k = 0 ; k < p ; k++ )
        {
          sum = sum + first[c][k]*second[k][d];
        }
         multiply[c][d] = sum;
        sum = 0;
      }
     }
    printf("Product of entered matrices:-\n");
    for ( c = 0 ; c < m ; c++ )
    {
     for ( d = 0 ; d < q ; d++ )
       printf("%d\t", multiply[c][d]);
      printf("\n");
    }
  }
  return 0;
}
```

Output:

```
Enter the elemtns of first matrix
1  2  0
0  1  1
2  0  1
Enter the number of rows and columns of second matrix
3
3
Enter the elements of second matrix
1  1  2
2  1  1
1  2  1
Product of entered matrices:-
5  3  4
3  3  2
3  4  5
```

Transpose of Matrix

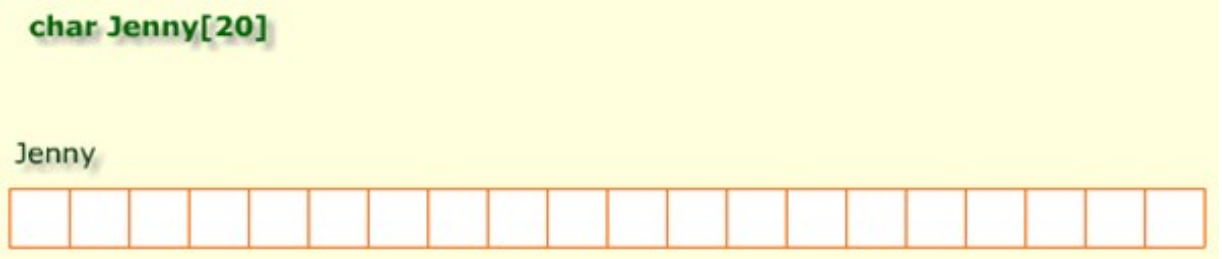
```
#include <stdio.h>
int main()
{
  int m, n, c, d, matrix[10][10], transpose[10][10];
  printf("Enter the number of rows and columns of matrix ");
  scanf("%d%d",&m,&n);
  printf("Enter the elements of matrix \n");
   for( c = 0 ; c < m ; c++ )
  {
    for( d = 0 ; d < n ; d++ )
    {
      scanf("%d",&matrix[c][d]);
    }
  }
   for( c = 0 ; c < m ; c++ )
  {
    for( d = 0 ; d < n ; d++ )
    {
      transpose[d][c] = matrix[c][d];
    }
  }
   printf("Transpose of entered matrix :-\n");
   for( c = 0 ; c < n ; c++ )
  {
    for( d = 0 ; d < m ; d++ )
    {
      printf("%d\t",transpose[c][d]);
    }
    printf("\n");
  }
   return 0;
}
```

Output:

```
Enter the number of rows and columns of matrix
2
3
Enter the elements of matrix
1  2  3
4  5  6
Transpose of entered matrix:-
1    4
2    5
3    6
```

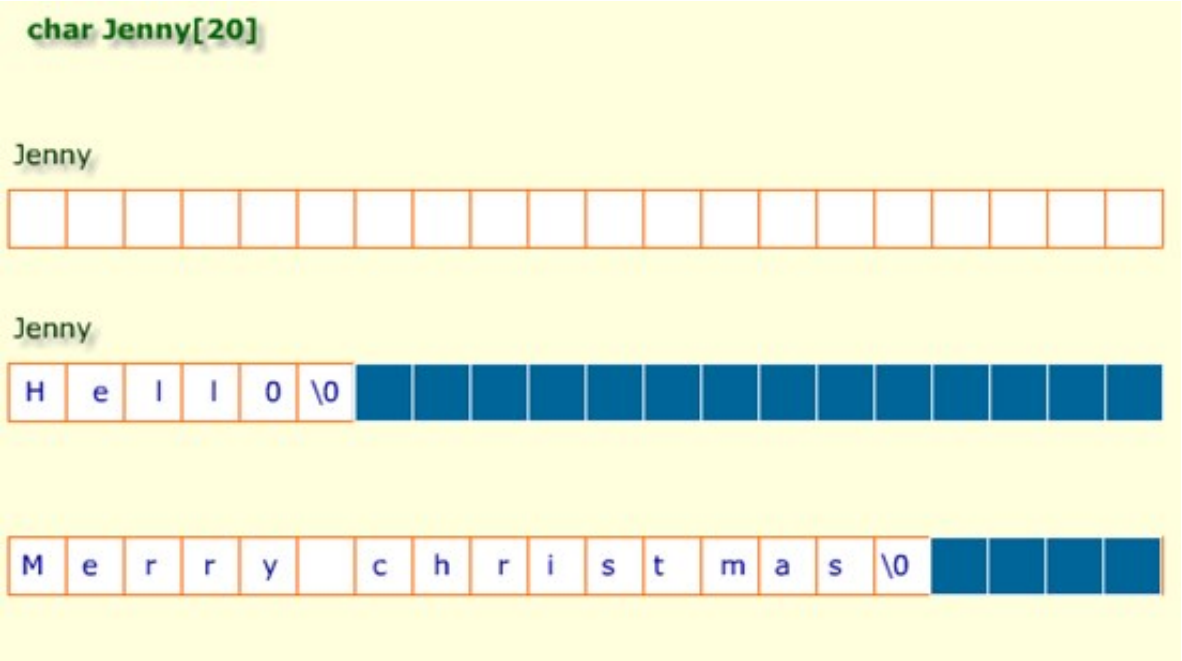
Null Terminated Strings as an Array of Characters

- C Standard Library implements a powerful string class, which is very useful to handle and manipulate strings of characters.
- However, because strings are in fact sequences of characters, we can represent them also as plain arrays of char elements.
- For example, the following array:
  - char jenny [20];
    - It's an array that can store up to 20 elements of type char.



String as an Array of Characters

- In the array, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences.
- For example, jenny could store at some point in a program either the sequence "Hello" or the sequence "Merry Christmas", since both are shorter than 20 characters.
- Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence:
  - The null character, whose literal constant can be written as '\0' (backslash, zero).
- Our array of 20 elements of type char, called jenny, can be represented storing the characters sequences "Hello" and "Merry Christmas" as:



Initialization of Null-terminated Character Sequences

- Because arrays of characters are ordinary arrays they follow all their same rules.
- For example:
  - If we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array:
    - char myword [ ] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
    - char myword [ ] = "Hello";

Representation Sparse Matrices

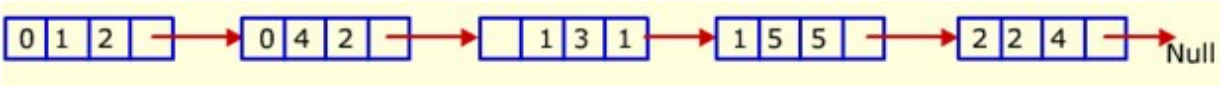
- A matrix is a two-dimensional data object made of m rows and n columns, therefore having m, n values.
- When m=n, we call it a square matrix.
- The most natural representation is to use two-dimensional array A [m][n] and access the element of i row and j column as A[i][j].
- If a large number of elements of the matrix are zero elements, then it is called a sparse matrix.
- Representing a sparse matrix by using a two-dimensional array leads to the wastage of a substantial amount of space.
- Therefore, an alternative representation must be used for sparse matrices.
- One such representation is to store only non- zero elements along with their row positions and column positions.
- That means representing every non-zero element by using triples (i, j, value), where i is a row position and j is a column position, and store these triples in a linear list.
- It is possible to arrange these triples in the increasing order of row indices, and for the same row index in the increasing order of column indices.
- Each triple (i, j, value) can be represented by using a node having four fields as shown in the following:

- A sparse matrix can be represented using a list of such nodes, one per non–zero element of the matrix.
- For example, consider the sparse matrix shown in figure.

0	2	0	0	2	0
0	0	0	1	0	5
0	0	4	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

A sparse matrix

- This matrix can be represented using the linked list shown in figure.



Example: Sparse matrix

```
* C program to determine if a given matrix is a sparse matrix.
 * Sparse martix has more zero elements than nonzero elements.
 */
#include <stdio.h>
 void main ()
{
   static int array[10][10];
   int i, j, m, n;
   int counter = 0;

   printf("Enter the order of the matix \n");
   scanf("%d %d", &m, &n);
   printf("Enter the co-efficients of the matix \n");
   for (i = 0; i < m; ++i)
   {
      for (j = 0; j < n; ++j)
      {
         scanf("%d", &array[i][j]);
         if (array[i][j] == 0)
         {
            ++counter;
         }
      }
   }
   if (counter > ((m * n) / 2))
   {
      printf("The given matrix is sparse matrix \n");
   }
   else
      printf("The given matrix is not a sparse matrix \n");
   printf("There are %d number of zeros", counter);
}
```

Output:

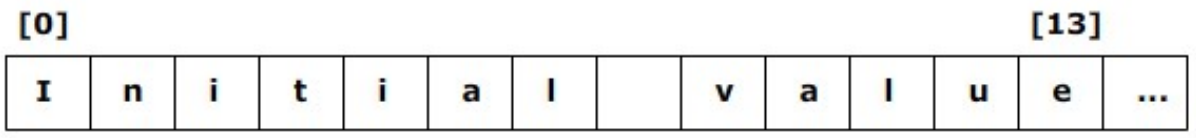
```
Enter the order of the matix
3 3
Enter the co-efficients of the matix
10 20 30
5 10 15
3 6 9
The given matrix is not a sparse matrix
There are 0 number of zeros

Enter the order of the matix
3 3
Enter the co-efficients of the matix
5 0 0
0 0 5
0 5 0
The given matrix is sparse matrix
There are 6 number of zeros
```

Standard Library String Functions - Strings

- C implements the string data structure using arrays of type char.
- The users have already used the string extensively.
  - printf(“This program is terminated!\n”);
  - #defineERR_Message “Error!!”
- Since string is an array, the declaration of a string is the same as declaring a char array.
  - char string_var[30];
  - char string_var[20] = “Initial value”;

Memory Storage for a String

- The string is always ended with a null character ‘\0’.
- The characters after the null character are ignored.
  - E.g., char str[20] = “Initial value”;



Arrays of Strings

- An array of strings is a two-dimensional array of characters in which each row is one string.
  - char names[People][Length];
  - char month[5][10] = {“January”, “February”, “March”, “April”, “May”};

Input/Output of a String

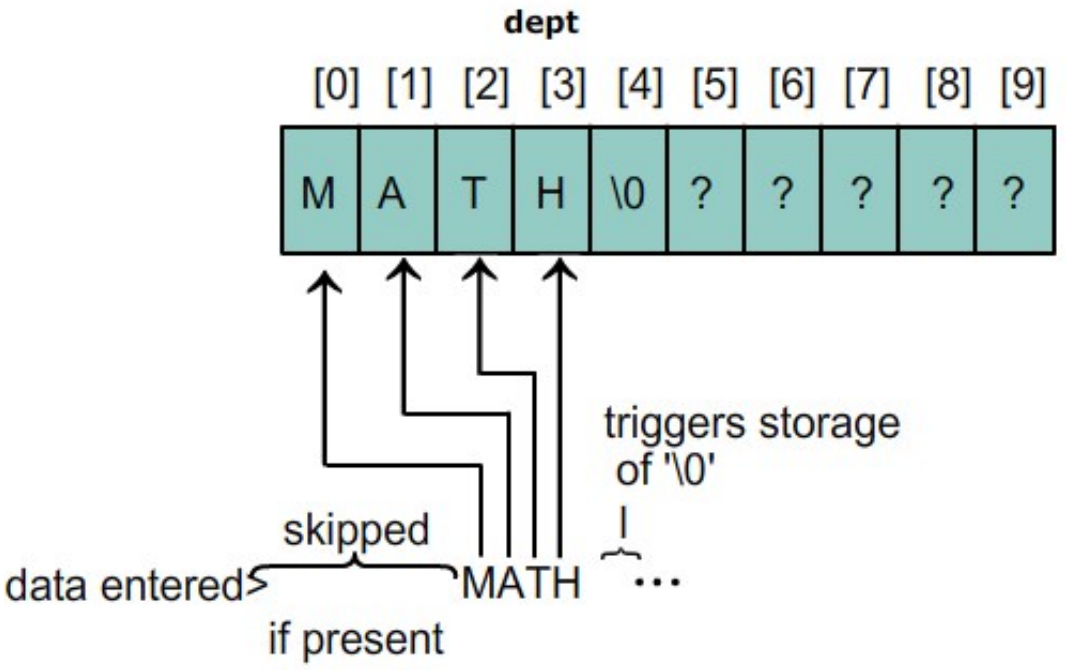
- The placeholder %s is used to represent string arguments in printf and scanf.
  - printf(“Topic: %s\n”, string_var);
- The string can be right-justified by placing a positive number in the placeholder.
  - printf(“%8s”, str);
- The string can be left-justified by placing a negative number in the placeholder.
  - Printf(“%-8s”, str);

An Example of Manipulating String with scanf and printf

```
#include <stdio.h>

#define STRING_LEN 10

int
 main(void)
{
            Char dept[STRING_LEN];
            int   COurse_num;
            Char days[STRING_LEN];
             int time;

             Printf("Enter department Code, Course number,days and");
             printf("time like this:\n> COSC 2060 MWF 1410\n>");
             Scanf("%s%d%s%d",dept,&Course_num,days,&time);
              Printf("%s %dmeets %s at %d\n",course_num,days,time);

             return(0);
}
```

Execution of scanf ("%s", dept);

- Whenever encountering a white space, the scanning stops and scanf places the null character at the end of the string.
  - E.g., if the user types “MATH 1234 TR 1800,” the string “MATH” along with ‘0’ is stored into dept.



String Library Functions

- The string can not be copied by the assignment operator ‘=’.
  - E.g, “str = “Test String”” is not valid.
- C provides string manipulating functions in the “string.h” library.

- The complete list of these functions can be found in Appendix B of the textbook.

Some String Functions from String.h

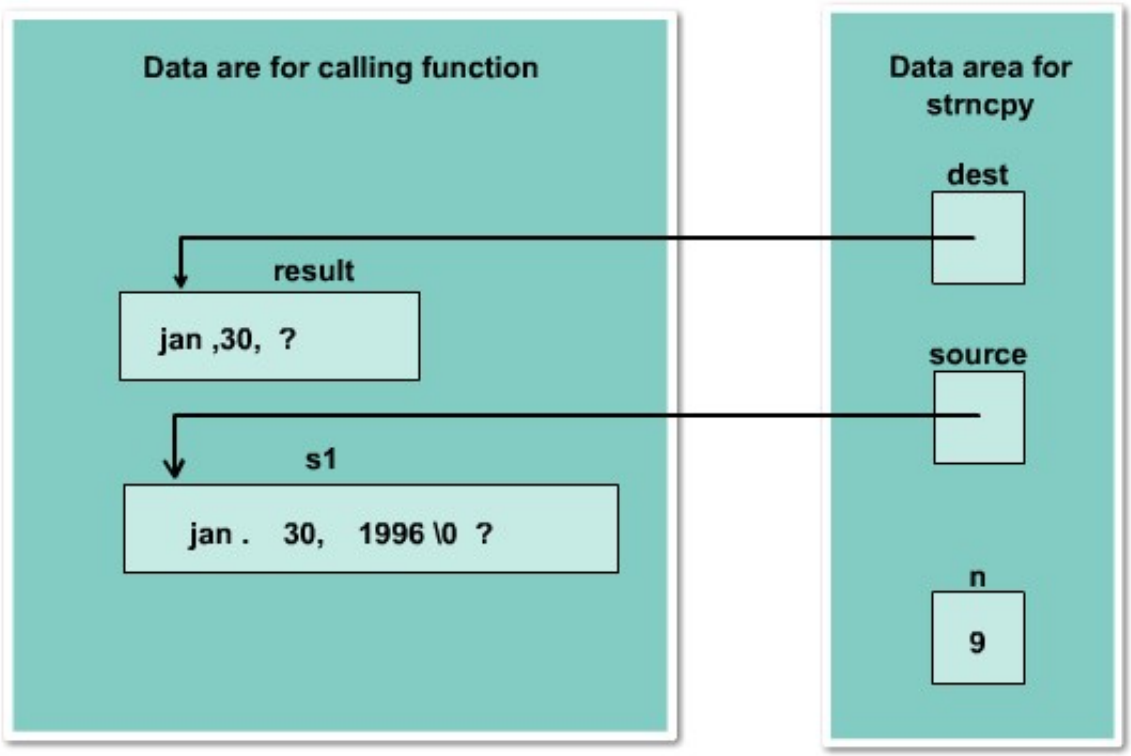
Function	Purpose	Example
strcpy	Makes a copy of a string	strcpy(s1, “Hi”);
strcat	Appends a string to the end of another string	strcat(s1, “more”);
strcmp	Compare two strings alphabetically	strcmp(s1, “Hu”);
strlen	Returns the number of characters in a string	strlen(“Hi”) returns 2.
strtok	Breaks a string into tokens by delimiters.	strtok(“Hi, Chao”, “ ,”);

Functions strcpy and strncpy

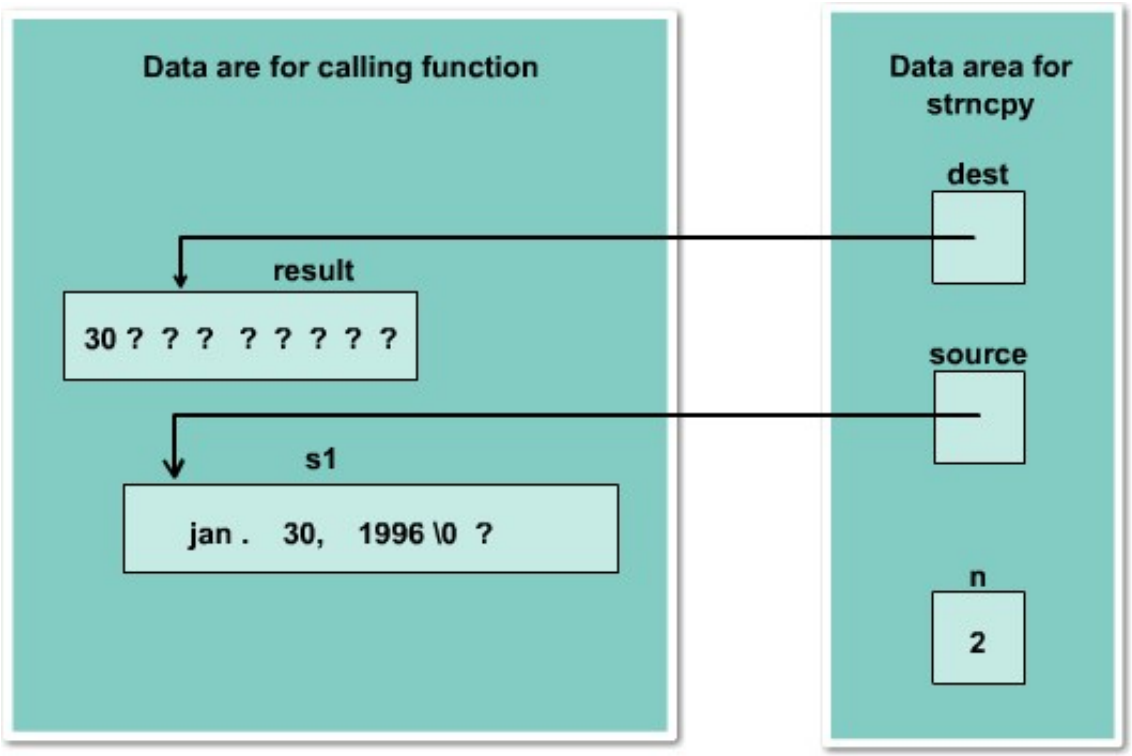
- Function strcpy copies the string in the second argument into the first argument.
  - E.g., strcpy(dest, “test string”);
  - The null character is appended at the end automatically.
  - If source string is longer than the destination string, the overflow characters may occupy the memory space used by other variables.
- Function strncpy copies the string by specifying the number of characters to copy.
- The users have to place the null character manually.
  - E.g., strncpy(dest, “test string”, 6); dest[6] = ‘\0’;
  - If source string is longer than the destination string, the overflow characters are discarded automatically.

Extracting Substring of a String

- We can use strncpy to extract substring of one string.
  - E.g., strncpy(result, s1, 9);



- E.g., strncpy(result, &s1[5], 2);



Functions strcat and strlen

- Functions strcat and strncat concatenate the fist string argument with the second string argument.
  - strcat(dest, “more..”);
  - strncat(dest, “more..”, 3);
- Function strlen is often used to check the length of a string (i.e., the number of characters before

the fist null character).

- E.g., dest[6] = “Hello”;
- strncat(dest, “more”, 5-strlen(dest));
- dest[5] = ‘\0’;

Distinction Between Characters and Strings

- The representation of a char (e.g., ‘Q’) and a string (e.g., “Q”) is essentially different.
  - A string is an array of characters ended with the null character.

String Comparison (1/2)

- Suppose there are two strings, str1 and str2.
  - The condition str1 < str2 compare the initial memory address of str1 and of str2.
- The comparison between two strings is done by comparing each corresponding character in them.
  - The characters are compared against the ASCII table.
  - “thrill” > “throw” since ‘i’ < ‘o’;
  - “joy” < joyous“;
- The standard string comparison uses the strcmp and strncmp functions.

String Comparison (2/2)

Relationship	Returned Value	Example
str1 < str2	Negative	“Hello” < “Hi”
str1 = str2	0	“Hi” = “Hi”
str1 > str2	Positive	“Hi” > “Hello”

- E.g., we can check if two strings are the same by
  - if(strcmp(str1, str2) != 0)
  - printf(“The two strings are different!”);

Input/Output of Characters and Strings

- The stdio library provides getchar function which gets the next character from the standard input.
  - “ch = getchar();” is the same as “scanf(“%c”, &ch);”
  - Similar functions are putchar, gets, puts.
- For IO from/to the file, the stdio library also provides corresponding functions.
  - getc: reads a character from a file.
  - Similar functions are putc, fgets, fputs.

UNIT - 6
Functions

Top-down Approach of Problem Solving

- There are three general approaches to writing a program:
  - Top down - In the top down approach one starts with the top-level routine and move down to the low level routine.
  - Bottom up - The bottom-up approach works in the opposite direction on begins with the specific routines, build them into progressively more complex structures, and end at the top-level routine.
  - Ad hoc - The ad hoc approach specifies no predetermined method. C as a structured language lends itself to the top down approach. The top down method can produce clean readable code that can easily be maintained

Top-down approach

- A top-down approach also helps one to clarify the overall structure and operation of the program before one code the low-level functions.
- The top down method starts with a general description and works towards specifics.
- In fact a good way to design a program is to define exactly what the program is going to do at the top level.
- Each entry in the list should contain only one functional unit.
- A functional unit can be thought of as a black box that performs a single task.

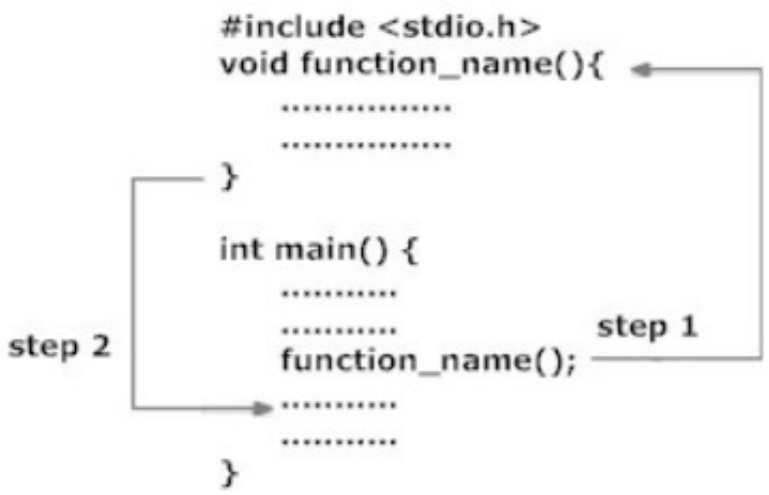
Modular Programming and Functions

Modular programming

- Modular programming is a style that adds structure and readability to the program code.
- It may not make much difference on small projects, but as one starts to work on something bigger it can make the code much easier to read and maintain.
- Structuring the code is a simple task of splitting the program into manageable chunks so that each part is self-contained.
- By creating these self-contained modules, one can focus on programming each part.
- Once one is satisfied that one bit is working, one can then move to the next module with the confidence, whatever one does elsewhere will not have an on effect on other modules that one has already written.

Functions

- What is a function?
  - A function is a named, independent section of C code that performs a specific task and optionally returns a value to the calling program.
  - A function is named. Each function has a unique name.
  - By using the name in another part of the program, one can execute the statements contained in the function.
  - This is known as calling the function.
  - A function can be called from within any other function.
  - A function is independent.
  - A function can perform its task without interference from or interfering with other parts of the program.
- General form of a function


#include <stdio.h>
void function_name(){
.....
.....
}
int main() {
.....
.....
function_name();
.....
.....
}
step 1
step 2

Standard C Library

- A collection of reusable functions (code for building data structures, code for performing math functions and scientific calculations, etc.,).
- Which will save C programmers time especially when working on large programming projects.
- The C Library is part of the ANSI (American National Standard Institute) for the C Language.
- The C programs can call on a large number of functions from the standard C library.
- These functions perform essential services such as input and output.
- They also provide efficient implementations of frequently used operations.
- Many macros and type definitions accompany these functions and help them to make better use of the standard C functions.

Standard Library of C functions

- C Standard library functions or simply C Library functions are inbuilt functions in C programming.
- Function prototype and data definitions of these functions are written in their respective header file.
- For example: If you want to use printf() function, the header file < stdio.h > should be included.

```
/* write printf() statement without including  header file,
this program will show error. */

printf(#include <stdio.h>

int main()

   {

/* If you "Hello World");

   }
```

- There is at least one function in any C program, i.e., the main() function (which is also a library function).
- This program is called at program starts.
- There are many library functions available in C programming to help the programmer to write a good efficient program.
- Suppose, you want to find the square root of a number.
- You can write your own piece of code to find square root but, this process is time consuming and

the code you have written may not be the most efficient process to find square root.

- But, in C programming you can find the square root by just using sqrt() function which is defined under header file "math.h".

Use of Library Function : To Find Square root

```
#include <stdio.h>
#include <math.h>
int main(){
float num,root;
printf("Enter a number to find square root.");
scanf("%f",&num);
root=sqrt(num);  /* Computes the square root of num and stores in root. */
printf("Square root of %.2f=%.2f",num,root);
return 0;
}
```

- Listed below are few of the standard C Libraries.
  - Stdio.h - Supports File Input/Output Operations.
  - Stdlib.h - Supports Miscellaneous declarations.
  - Math.h- Supports Definitions used for mathematical calculations.
  - String.h - Supports string functions.
  - Time.h- Supports system time functions.
  - Ctype.h - Header file "ctype.h" includes numerous standard library functions to handle characters (especially test characters).

Example Programs of frequently used standard C Libraries

```
#include <stdio.h>
int main()
{
   int this_is_a_number;
   printf( "Please enter a number: " );
   scanf( "%d", &this_is_a_number );
   printf( "You entered %d", this_is_a_number );
   getchar();
   return 0;
}
```

Prototype of a Function

- Function Prototype:return_typefunction_name(arg-type name-1,...,arg-type name-n);
- A function prototype provides the compiler with the description of a function that will be defined at a later point in the program.
- The prototype includes a return type indicating the type of variable that the function will return;it also includes the function name, which should describe what the function does.
- The prototype also contains the variable types of the arguments (arg type) that will be passed to the function.
- Optionally, it can contain the names of the variable that will be passed.
- A prototype should always end with a semicolon.
  - double squared(double number);
  - void print_report(intreport_number);
  - int get_menu_choice(void);

Function Definition

- A function definition is the actual function.
- The definition contains the code that will be executed.
- The first line of a function definition, called the function header, should be identical to the function prototype, with the exception of the semicolon.
- A function header shouldn′t end with a semicolon.
- In addition, although the argument variable names the optional in the prototype, they must be included in the function header.
- Following the header is the function body, containing the statements that the function will perform.
- The function body should start with an opening bracket and end with a closing bracket.
- If the function return type is anything other than void, a return statement should be included, returning a value matching the return type.
  - return_typefunction_name(arg-type name-1,...,arg-type name-n)
  - {
  - /* statements; */
  - }

- The following Program demonstrates prototype of a function

```
#include <stdio.h>
   int sum (int, int);
    int  main (void)
   {
      int total;
      total = sum (2, 3);
      printf ("Total is %d\n", total);
      return 0;
   }
   int
   sum (int a, int b)
   {
      return a + b;
   }
```

Output:

```
Total is
5
```

Formal Parameter List

Parameter

- A parameter is a special kind of variable, used in a subroutine to refer to one of the pieces of data provided as input to the subroutine.
- These pieces of data are called arguments.
- Parameter Means Values Supplied to Function so that Function can utilize these Values.
- Parameters are Simply Variables.
- Difference between Normal Variable and Parameter is that “These Arguments are Defined at the time of Calling Function“.

Formal Parameter

- Parameter Written in Function Definition is Called “Formal Parameter.
- Although formal parameters are always variables actual parameters do not have to be variables.

Example

```
void main()
{
int num1;
display(num1);
}
void display(int para1)
{
-----------
-----------
}
```

- Para1 is called ‘Formal Parameter’.

Actual Parameter

- Parameter Written In Function Call is Called “Actual Parameter”.
- One can use numbers, expressions, or even function calls as actual parameters.

Example

```
void main()
{
int num1;
display(num1);
}
void display(int para1)
{
-----------
-----------
}
```

- num1 is ‘Actual Parameter’.

Parameter list

- A function is declared in the following manner:
  - return-type function-name(parameter-list,...) { body... }
- Return-type is the variable type that the function returns.
- This cannot be an array type or a function type.
- If not given, then int is assumed.
- Function-name is the name of the function.

The Function Return Type

- The function return type specifies the data type that the function returns to the calling program.
- The return type can be any of C′s data types: char, int , long , float , or double .
- One can also define a function that doesn′t return a value by using a return type of void.
- Given below are few examples:
  - int func1(...) /* Returns a type int. */
  - float func2(...) /* Returns a type float. */
  - void func3(...)

Return Type – Program

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
  /* local variable declaration */
  int result;
   if (num1 > num2)
     result = num1;
  else
     result = num2;
   return result;
}
```

Calling Functions

- There are two ways to call a function.
- Any function can be called by simply using its name and argument list alone in a statement, as in the following example.
- If the function has a return value, it is discarded.
  - wait(12);
- The second method can be used only with functions that have a return value.
- Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used.
- An expression with a return value used as the right side of an assignment statement.
- In the following example, half _of() is a parameter of a function:printf("Half of %d is %d.", x, half _of(x));
- First, the function half _of() is called with the value of x, and then printf() is called using the values x and half_of(x).

- In this second example, multiple functions are being used in an expression: y = half_of(x) + half_of(z);
- Although half_of() is used twice, the second call could have been any other function.
- The following code shows the same statement, but not all on one line:
  - a = half_of(x);
  - b = half_of(z);
  - y = a + b;

Example Program

```
#include <stdio.h>
 /* function declaration */
int max(int num1, int num2);
 int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
    /* calling a function to get max value */
   ret = max(a, b);
   printf( "Max value is : %d\n", ret );
   return 0;
}
```

Block structure

- A program is made up of one or more functions, with one of these being main ().
- Function is a self-contained block of program that performs a particular task.
- This is the structure of function

```
main()
{
Message();
}
Message()
{
printf("Function Block Structure");
return;
}
```

- Example of Block Structure of program: How to create blocks using {} and their significance.

```
#include <stdio.h>
int main()
{
 int x;
 x = 0;
 do
{
   printf( "Hello, world!\n" );
 }
    while ( x != 0 );
     getchar();
}
```

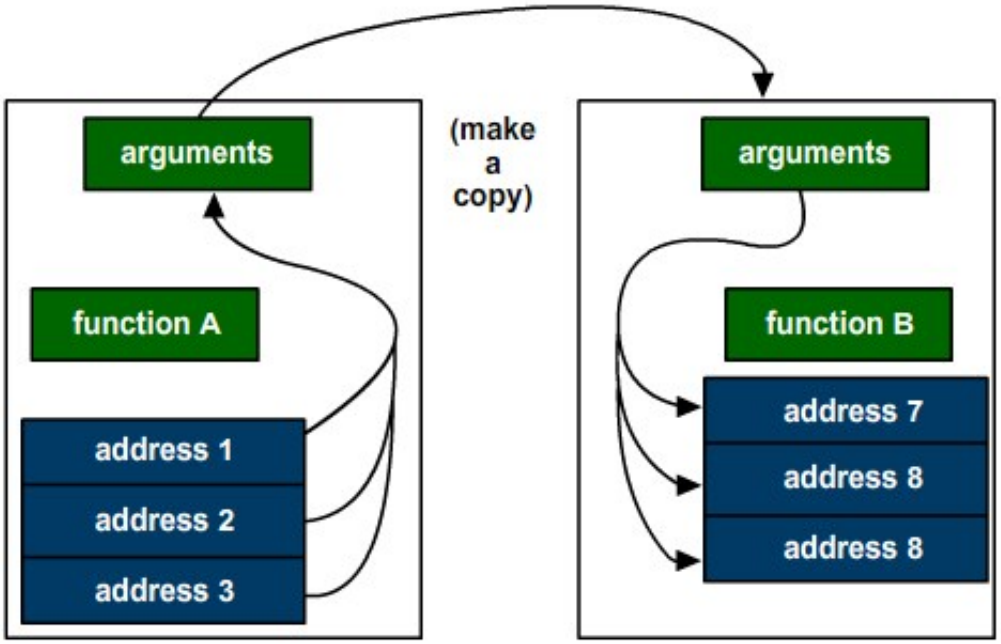
Passing Arguments to a Function

- To pass arguments to a function, list them in parentheses following function name.
- The number of arguments and the type of each argument must match the parameters in the function header and prototype.
- For example, if a function is defined to take two type int arguments, one must pass it exactly two int arguments--no more, no less--and no other type.
- If one tries to pass a function an incorrect number and/or type argument, the compiler will detect it, based on the information in the function prototype.
- If the function takes multiple arguments, the arguments listed in the function call are assigned to the function parameters in order.
- The first argument to the first parameter, the second argument to the second parameter, and so on.
- Multiple arguments are assigned to function parameters in order.
- Each argument can be any valid C expression: a constant, a variable, a mathematical or logical expression, or even another function (one with a return value).
- For example, if half(), square(), and third() are all functions with return values, one could write as follows:
  - x = half(third(square(half(y))));
    - The program first calls half(), passing it y as an argument.
- When execution returns from half(), the program calls square(), passing half()'s return value as an argument.
- Next, third() is called with square()'s return value as the argument.
- Then, half() is called again, this time with third()'s return value as an argument.
- Finally, half()'s return value is assigned to the variable x.
- The following is an equivalent piece of code:
  - a = half(y);
  - b = square(a);
  - c = third(b);
  - x = half(c);

Call by Reference

In this method the address of an argument is copied into the parameter.

- Inside the subroutine the address is used to access the actual argument used in the call.
- This means that the change made to the parameter affect the variable used to call the subroutine.
- The address of x and y are passed as the parameter to the function swap().



- The following Program demonstrates a call by Reference.

```
#include<stdio.h>

void interchange(int *num1,int *num2)
{
   int temp;
   temp  = *num1;
   *num1 = *num2;
   *num2 = temp;
}

int main() {

   int num1=50,num2=70;
   interchange(&num1,&num2);

   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);

   return(0);
}
```

Output :

```
Number 1 : 70
Number 2 : 50
```

Call by Value

- This method copies value of the argument into the formal parameter of the subroutine.
- Therefore changes made to the parameters of the subroutines have no effect on the variable used to call it.
- The value of t is passed as the parameter to the function sqr().
- The following Program demonstrates call by value.

```
#include<stdio.h>

void interchange(int number1,int number2)
{
   int temp;
   temp = number1;
   number1 = number2;
   number2 = temp;
}

int main() {

   int num1=50,num2=70;
   interchange(num1,num2);

   printf("\nNumber 1 : %d",num1);
   printf("\nNumber 2 : %d",num2);

   return(0);
}
```

Output :

```
Number 1 : 50
Number 2 : 70
```

Recursive Functions

- The term recursion refers to a situation in which a function calls itself either directly or indirectly.
- Indirect recursion occurs when one function calls another function that then calls the first function.
- C allows recursive functions, and they can be useful in some situations.
- For example, recursion can be used to calculate the factorial of a number as shown below.

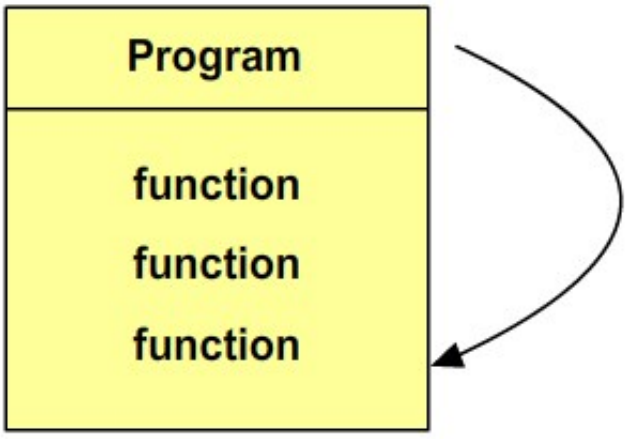
```
#include <stdio.h>

int factorial(unsigned int i)
{
  if(i <= 1)
  {
    return 1;
  }
  return i * factorial(i - 1);
}
int  main()
{
   int i = 15;
   printf("Factorial of %d is %d\n", i, factorial(i));
   return 0;
}
```

- The following Program demonstrates recursive function.

```
#include <stdio.h>
int fibonaci(int i)
{
  if(i == 0)
  {
    return 0;
  }
  if(i == 1)
  {
    return 1;
  }
  return fibonaci(i-1) + fibonaci(i-2);
}
int  main()
{
   int i;
   for (i = 0; i < 10; i++)
   {
     printf("%d\t%n", fibonaci(i));
   }
   return 0;
}
```

Arrays as Function Arguments

- An argument is a value that the calling program passes to a function.



- It can be an int, a float, or any other simple data type, but it must be a single numerical value.
- It can be a single array element, but it can't be an entire array.
- One can have a pointer to an array, and that pointer is a single numeric value (the address of the array's first element).
- If one passes that value to a function, the function knows the address of the array and can access the array elements using pointer notation.
  - int largest(int x[ ], int y)
- The largest() is a function that returns an int to the calling program; its second argument is an int represented by the parameter y.
- The only thing new is the first parameter, intx[ ], which indicates that the first argument is a pointer to type int, represented by parameter x.
- Write the function declaration and header as shown below:int largest(int *x, int y);
- This is equivalent to the first form; both intx [ ] and int *x mean "pointer to int."
- The first form might be preferable.
- The parameter represents a pointer to an array.
- Of course, the pointer doesn't know that it points to an array, but the function uses it that way.

UNIT - 7

Storage Classes

Scope of a Variable

- The scope of a variable determines over what part(s) of the program a variable is actually available for use(active).
- Longevity: it refers to the period during which a variables retains a given value during execution of a program(alive).
- Local (internal) variables: are those which are declared within a particular function.
- Global (external) variables: are those which are declared outside any function.

Scope of a Declaration

- Scope of a declaration is the region of C program text over which that declaration is active.
  - Top-level identifiers – Extends from declaration point to end of file.
  - Formal parameters in functions – Extends from declaration point to end of function body.
  - Block/function (local) identifiers – Extends from declaration point to end of block/function.

Extent

- The extent of an object is the period of time that its storage is allocated.
- An object is said to have static extent when it is allocated storage at or before the beginning of program execution and the storage remains allocated until program termination.
- All functions have static extent, as do all variables declared in top-level declarations and variables declared with the static qualifier.
- Formal parameters and variables declared at the beginning of a block or function have local extent (dynamic, de-allocated on block/function exit).

External variable or function

- A top-level identifier is treated just like a static extent object in the file containing its definition, but is also exported to the linker, and if the same identifier is declared in another file, the linker will ensure the two files reference the same object (variable or function).
- Thus, top-level identifiers and functions are external by default.
- The convention is to use the extern qualifier in all places that are simply declaring an identifier that is defined (allocated) in another file, and to omit the qualifier at the single point where the variable is actually defined (allocated space).
- Thus external is the same as “global” or “public”.
- A static variable or function: has only file scope, is not external.
- Thus static is the same as “private.”

Examples

- Variables, top-level:

```
int  x = 3;      //defining, exported (external by default) - static extent
extern  int w;   //declaring, imported (variable is defined elsewhere)
static  int y;   //not exported/external (file scope) - static extent
Functions:
int f(int xx){   //exported (external by default) - static extent

intzz;           //local (function scope) - local extent

...

}

staticint g(intyy){ //not exported (file scope)) - static extent

...

}
```

Storage Classes

- A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.
- There are following storage classes which can be used in a C Program
  - Auto,
  - Extern,
  - Static,
  - Register.

Automatic variables

- Are declared inside a function in which they are to be utilized.
- Are declared using a keyword auto. eg. auto int number;
- Are created when the function is called and destroyed automatically when the function is exited.
- These variables are therefore private (local) to the function in which they are declared.
- Variables declared inside a function without storage class specification is, by default, an automatic variable.
- Auto is the default storage class for all local variables.

```
{
int Count;
auto int Month;
}
```

- The example above defines two variables with the same storage class.
- Auto can only be used within functions, i.e. local variables.

Example program

```
int main()
{
int m=1000;
function2();
printf("%d\n",m);
}
function1()
{
int m = 10;
printf("%d\n",m);
}
function2()
{
int m = 100;
function1();
printf("%d\n",m);
}
```

Output

```
10
100
1000
```

About Auto Variables

- Any variable local to main will normally live throughout the whole program, although it is active only in main.
- During recursion, the nested variables are unique auto variables.
- Automatic variables can also be defined within blocks.
- In that case, they are meaningful only, inside the blocks where they are declared.
- If automatic variables are not initialized - will contain garbage.
- The features of automatic variables are
  - Storage – memory.
  - Default initial value - an unpredictable value, which is often a garbage value.
  - Scope - local to the block in which the variable is defined.
  - Life - till the control remains within the block variable is defined.

External Variables

- These variables are declared outside any function.
- These variables are active and alive throughout the entire program.
- Also known as global variables and default value is zero.

- Unlike local variables they can be accessed by any function in the program.
- In case local variable and global variable have the same name, the local variable will have precedence over the global one.
- Sometimes the keyword extern is used to declare these variables.
- Extern is used to give a reference of a global variable that is visible to all the program files.
- It is visible only from the point of declaration to the end of the program.
- The extern variable cannot be initialized, as all it does is, point the variable name at a storage location that has been previously defined.
- When a programmer has multiple files and he defines a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function.
- Extern is used to declare a global variable or function in another file.

External Declaration

```
int main()
{
 y=5;
  . . .
  . . .
}
int y;
func1()
{
 y=y+1
}
```

- As far as main is concerned, y is not defined.
- So compiler will issue an error message.
- There are two way out at this point
  - Define y before main.
  - Declare y with the storage class extern in main before using it.

External Declaration(Examples)

```
int main()
{
 extern int y;
 . . .
 . . .
}
func1()
{
 extern int y;
 . . .
 . . .
}
int y;
```

- Note that extern declaration does not allocate storage space for variables.

External Variable (Examples)

- extern int a=1, b=2;
  - External variable never disappear.
  - Because exist throughout the execution life of the program, they can be used to transmit value across functions.
  - They may be, hidden if the identifier is redefined.
  - All functions have external storage class.
  - This means that the key word extern can be used in function definitions and in function prototypes.
- For example;
  - extern double sin(double);
- In a function prototype, for the sin() function; its function definition – we can write;

```
extern double sin(double)

{

------

------

------

-------

-------

}
```

- An extern within a function provides the type of information just to that one function.
- The extern declaration does not allocate storage space for variables.
- The features of external storage class variable are as follows:
  - Storage — memory.
  - Default initial value — zero.
  - Scope — global.
  - Life — as long as the program execution does not end.

Example

File 1: main.c

```
int count=5;
main()
  {
   write_extern ();
  }
```

File 2: write.c

```
void write_extern(void);
extern int count;
  {
    printf("count is %i\n", count);
  }
```

- Here extern keyword is being used to declare count in another file.
- Now compile these two files.

- This file produce write program which can be executed to produce result.
- Count in ‘main.c’ will have a value of 5. If main.c changes the value of count - write.c will see the new value.
- When the function references the variable count, it will be referencing only its local variable, not the global one.

Global Variable Example

```
int x;
int main()
 {
  x=10;
printf("x=%d\n",x);
printf("x=%d\n",fun1());
printf("x=%d\n",fun2());
printf("x=%d\n",fun3());
 }
int fun1()
 { x=x+10;
   return(x);
 }
int fun2()
 { int x
   x=1;
   return(x);
 }
int fun3()
 {
   x=x+10;
   return(x);
 }
```

Output

```
x=10
x=20
x=1
x=11
```

- Once a variable has been declared global any function can use it and change its value.
- The subsequent functions can then reference only that new value.

Static Variables

- The value of static variables persists until the end of the program.
- It is declared using the keyword static like
  - static int x;
  - static float y;

- It may be of external or internal type depending on the place of the declaration.
- Static variables are initialized only once, when the program is compiled.
- Static is the default storage class for global variables.
- The two variables below (count and road) both have a static storage class.

```
static int Count;

int Road;

      {

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

      }
```

- Static variables can be 'seen' within all functions in this source file.
- At link time, the static variables defined here will not be seen by the object modules that are brought in.
- Static can also be defined within a function.
- If this is done, the variable is initialized at run time, but is not reinitialized, when the function is called.
- Inside a function, static variable retains its value during various calls.

Example

```
static int Count;
int Road;
      {
printf("%d\n", Road);
      }
  void func(void);
  static count=10; /* Global variable - static is the default */
  main()
  {
    while (count--)
    {
func();
    }
  }
  void func( void )
  {
    static int i = 5;
    i++;
printf("i is %d and count is %d\n", i, count);
  }
```

- This will produce following result

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

- NOTE: Here keyword 'void' means function does not return anything and it does not take any parameter.

Static Function

- Static declaration can also be used to control the scope of a function.
- If one wants a particular function to be accessible only to the functions in the file in which it is defined and not to any function in other files, declare the function to be static. eg.

```
static int power(int x inty)
    {
     . . .
     . . .
    }
```

Internal Static Variable

- Are those which are declared inside a function?
- Scope of Internal static variables extends up to the end of the program in which they are defined.
- Internal static variables are almost same as auto variable except they remain in existence (alive) throughout the remainder of the program.
- Internal static variables can be used to retain values between function calls.

Examples (Internal Static)

- Internal static variable can be used to count the number of calls made to function. eg.

```
int main()
{
inti;
  for(i =1; i<=3; i++)
     stat();
 }
void stat()
{
  static int x=0;
  x = x+1;
printf("x = %d\n",x);
}
```

Output

```
x=1
x=2
x=3
```

External Static Variables

- An external static variable is declared outside of all functions and is available to all the functions in the program.
- An external static variable seems similar simple external variable but their difference is that static external variable is available only within the file where it is defined while simple external variable can be accessed by other files.

Register Variable

- These variables are stored in one of the machine's register instead of RAM and are declared using register keyword.
  - Eg. register int count;
- This means that the variable has a maximum size equal to the register size (usually one word) and cannot have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  Miles;
}
```

- Since register access are much faster than a memory access keeping frequently accessed variables in the register lead to faster execution of program.
- Register should only be used for variables that require quick access - such as counters.
- It should also be noted that defining 'register' does not mean that the variable will be stored in a register.

  It means that it MIGHT be stored in a register - depending on hardware and implementation

- restrictions.
- Since only few variables can be placed in the register, it is important to carefully select the variables for this purpose.
- However, C will automatically convert register variables into non-register variables once the limit is reached.
- Don't try to declare a global variable as register. Because the register will be occupied during the lifetime of the program.

Multifile Programs and Extern Variables

file1.c

```
int main()
{
   extern int m;
inti
   . . .
   . . .
}
function1()
{
int j;
    . . .
    . . .
}
```

File2.c

```
int m;
function2()
{
inti
   . . .
   . . .
}
function3()
{
int count;
    . . .
    . . .
}
```

Multi Files - Static Variables

Example

File1.c

```
static int i = 10;
void main()
{
   i is accessible
}
void function ()
{
   i is accessible
}
```

File2.c

```
void function2()
{
   static i of file1.c is not
   accessible here
}
void function2 ()
{
   static i of file1.cis not
   accessible here
}
```

UNIT 8

Structures and Unions

Introduction: Data Types

- C programming language which has the ability to divide the data into different types.
- The type of a variable determines the kind of values it may take on.
- The various data types are
  - Simple Data type: Integer, Real, Void, Char.
  - Structured Data type: Array, Strings.
  - User Defined Data type: Enum, Structures, Unions.

Structure Data Type

- A structure is a user defined data type that groups logically related data items of different data types into a single unit.
- All the elements of a structure are stored at contiguous memory locations.
- A variable of structure type can store multiple data items of different data types under the one name.
- As the data of employee in company that is name, Employee ID, salary, address, phone number is stored in structure data type employee_detail.

Defining Structure

- A structure has to defined, before it can used.
- The syntax of defining a structure is

```
struct<struct_name>
{
<data_type><variable_name>;
<data_type><variable_name>;
.....
<data_type><variable_name>;
 };
```

Example of Structure

- The structure of Employee is declared as

```
struct employee
{
in temp_id;
char name[20];
float salary;
char address[50];
int dept_no;
int age;
};
```

Application of structure

- Structure is used in database management to maintain data about books in library, items in store, employees in an organization, financial accounting transaction in company.
- Structure is used in C programming for following purposes
  - Clearing screen.
  - Adjusting cursor position.
  - Drawing any graphics shape on the screen.
  - Receiving a key from the keyboard.
  - Finding out the list of equipment attached to the computer.
  - Changing the size of the cursor.
  - Formatting a floppy.
  - Hiding a file from the directory.
  - Displaying the directory of a disk.
  - Checking the memory size.
  - Sending the output to printer.
  - Interacting with the mouse.

Structure Variables

- A structure is a collection of one or more variables grouped under a single name for easy manipulation.
- The variables in a structure, unlike those in an array, can be of different variable types.
- A structure can contain any of C′s data types, including arrays and other structures.
- Each variable within a structure is called a member of the structure.

Declaring a Structure Variable

- A structure has to declared, after the body of structure has defined.
- The syntax of declaring a structure is
  - struct< struct_name > < variable_name >;
- The example to declare the variable for defined structure “employee”.
  - struct employee e1;
- Here e1 variable contains 6 members that are defined in structure.

Structure Variables – Program

```
#include <stdio.h>
struct student {
   char firstName[20];
   char lastName[20];
   char SSN[10];
   float gpa;
};
main()
{
 struct student student_a;
 strcpy(student_a.firstName, "Ram");
 strcpy(student_a.lastName, "Kumar");
 strcpy(student_a.SSN, "2333234" );
 student_a.gpa = 2009.20;
 printf( "First Name: %s\n", student_a.firstName );
 printf( "Last Name: %s\n", student_a.lastName );
 printf( "SNN : %s\n", student_a.SSN );
 printf( "GPA : %f\n", student_a.gpa );
}
```

Initialization - Initializing Structure Members

- The members of individual structure variable are initializing one by one or in a single statement.
- The example to initialize a structure variable is

```
1. struct employee e1 = {1, "Hemant",12000,
                "3  vikas colony new delhi",10, 35);

   Or

2. e1.emp_id=1;
   e1.dept_no=1
   strcpy (e1.name,"Hemant");
   e1.age=35;
   e1.salary=12000;
   strcypy(e1.address,"3  vikas colony new delhi");
```

Accessing Structure Members

- The structure members cannot be directly accessed in the expression.
- They are accessed by using the name of structure variable followed by a dot and then the name of member variable.
- The method used to access the structure variables are e1.emp_id, e1.name, e1.salary, e1.address, e1.dept_no, e1.age.
- The data within the structure is stored and printed by this method using scanf and printf statement in c program.
- A structure named coord that contains both the x and y values of a screen location is as follows:

```
struct coord
  {
       int x;
       int y;
  };
```

- The struct keyword, which identifies the beginning of a structure definition, must be followed immediately by the structure name, or tag (which follows the same rules as other C variable names).
- Within the braces following the structure name is a list of the structure's member variables.

- The variable type and name must be provided for each member.

```
struct coord
  {
       int x;
       int y;
  }first, second;
```

- The above statements define a structure type named coord that contains two integer variables, x and y.
- They do not, however, actually create any instances of the structure coord.
- In other words, they don't declare (set aside storage for) any structures.
- There are two ways to declare structures.
- One is to follow the structure definition with a list of one or more variable names.
- These statements define the structure type coord and declare two structures, first and second, of type coord.
- First and second are each instances of type coord; first contains two integer members named x and y, and so does second.
- This method of declaring structures combines the declaration with the definition.
- The second method is to declare structure variables at a different location in the source code from the definition.
- The statements also declare two instances of type coord:
- The following program shows the structure for recording students.

```
struct student{
    char name[50];
    int roll;
    float marks;
};
int main(){
    struct student s[10];
    int i;
    printf("Enter information of students:\n");
    for(i=0;i<10;++i)
    {
        s[i].roll=i+1;
        printf("\nFor roll number %d\n",s[i].roll);
        printf("Enter name: ");
        scanf("%s",s[i].name);
        printf("Enter marks: ");
        scanf("%f",&s[i].marks);
        printf("\n");
    }
    printf("Displaying information of students:\n\n");
    for(i=0;i<10;++i)
    {
     printf("\nInformation for roll number %d:\n",i+1);
     printf("Name: ");
     puts(s[i].name);
     printf("Marks: %.1f",s[i].marks);
    }
   return 0;
}
```

Structure Assignment

- The value of one structure variable is assigned to another variable of same type using assignment statement.
- If the e1 and e2 are structure variables of type employee then the statement e1 = e2; assign value of structure variable e2 to e1.
- The value of each member of e2 is assigned to corresponding members of e1.
- Individual structure members can be used like other variables of the same type.
- Structure members are accessed using the structure member operator (.), also called the dot operator, between the structure name and the member name.
- Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, the code is as follows:
  - first.x = 50;
  - first.y = 100;
- To display the screen locations stored in the structure second, the code is as follows: printf("%d,%d", second.x, second.y);
- What is the advantage of using structures rather than individual variables.
- One major advantage is that a programmer can copy information between structures of the same type with a simple equation statement.
- Continuing with the preceding example, the statement:
  - first = second; is equivalent to this statement as follows,
    - first.x = second.x;
    - first.y = second.y.

Program to Implement the Structure

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

struct student
{
     int id;
     char name[20];
     float percentage;
} record;

int main()
{

    record.id=1;
    strcpy(record.name, "Raju");
    record.percentage = 86.5;
    printf(" Id is: %d \n", record.id);
    printf(" Name is: %s \n", record.name);
    printf(" Percentage is: %f \n", record.percentage);
    return 0;
}
```

Structures within Structures

- C language allows a variable of structure type to be a member of some other structure type.
- The syntax to define the structure within structure is

```
struct<struct_name>
{
<data_type><variable_name>;
struct<struct_name>
    { <data_type><variable_name>;
    ……..}<struct_variable>;
<data_type><variable_name>;
};
```

Example of Structure within Structure

```
#include <stdio.h>

struct Employee
{
  char ename[20];
  int ssn;
  float salary;
  struct date
     {
     int date;
     int month;
     int year;
     }doj;
}emp = {"Pritesh",1000,1000.50,{22,6,1990}};

int main(int argc, char *argv[])
{
printf("\nEmployee Name   : %s",emp.ename);
printf("\nEmployee SSN    : %d",emp.ssn);
printf("\nEmployee Salary : %f",emp.salary);
printf("\nEmployee DOJ    : %d/%d/%d", \
       emp.doj.date,emp.doj.month,emp.doj.year);

return 0;
}
```

Accessing Structures within Structures

- The data member of structure within structure is accessed by using two period (.) symbols.
- The syntax to access the structure within structure is
  - struct _var. nested_struct_var. struct_member;
- For Example:-
  - e1.doj.day;
  - e1.doj.month;
  - e1.doj.year;

Nested Structures

- In the example a defined structure has been dealt with, that can hold the two coordinates required for a single point.
- One needs two such structures to define a rectangle.
- One can define a structure as follows (assuming, of course, that one has already defined the type coord structure):

```
struct rectanlge
   {
        struct coord topleft;
        struct coord bottomright;
    };
```

- This statement defines a structure of type rectangle that contains two structures of type coord.
- These two type coord structures are named topleft and bottomrt.
- The preceding statement defines only the type rectangle structure.
- To declare a structure, one must include a statement such as:
  - struct rectangle mybox;
- One could have combined the definition and declaration, as he did before for the type coord:

```
struct rectanlge
   {
        struct coord topleft;
        struct coord bottomright;
    }mybox;
```

- To access the actual data locations (the type int members), one must apply the member operator (.) twice.
- Thus, the expression:
  - mybox.topleft.x
- Refers to the x member of the topleft member of the type rectangle structure named mybox.

- To define a rectangle with coordinates (0, 10), (100, 200), one can write.

```
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomright.x = 100;
mybox.bottomright.y = 200;
```

Structures and Functions

- In C, structure can be passed to functions by two methods:
  - Passing by value (passing actual value as argument).
  - Passing by reference (passing address of an argument).

Passing Structure by Value

- A structure variable can be passed to the function as an argument as normal variable.
- If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.
- Write a C program to create a structure student, containing name and roll.
- Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

```
#include <stdio.h>
struct student
{
   char name[50];
   int roll;
};
void Display(struct student stu);
/* function prototype should be below to the structure
       declaration otherwise compiler shows error */
int main(){
struct student s1;
printf("Enter student's name: ");
scanf("%s",&s1.name);
printf("Enter roll number:");
scanf("%d",&s1.roll);
   Display(s1);  // passing structure variable s1 as argument
   return 0;
}
void Display(struct student stu){
printf("Output\nName: %s",stu.name);
printf("\nRoll: %d",stu.roll);
}
```

Output

```
Enter student's name: Kevin Amla
Enter roll number: 149
Output
Name: Kevin Amla
Roll: 149
```

Passing Structure by Reference

- The address location of structure variable is passed to function while passing it by reference.
- If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.
- Write a C program to add two distances(feet-inch system) entered by user.
- To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by reference and display the result in main function without returning it.

```
#include <stdio.h>
struct distance
{
   int feet;
   float inch;
};
void Add(struct distance d1,struct distance d2, struct distance *d3);
int main()
{
struct distance dist1, dist2, dist3;
printf("First distance\n");
printf("Enter feet: ");
scanf("%d",&dist1.feet);
printf("Enter inch: ");
scanf("%f",&dist1.inch);
printf("Second distance\n");
printf("Enter feet: ");
scanf("%d",&dist2.feet);
printf("Enter inch: ");
scanf("%f",&dist2.inch);
Add(dist1, dist2, &dist3);
/*passing structure variables dist1 and dist2
 by value whereas passing structure variable dist3 by reference */
printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);
   return 0;
}
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
/* Adding distances d1 and d2 and storing it in d3 */
    d3->feet=d1.feet+d2.feet;
    d3->inch=d1.inch+d2.inch;
if (d3->inch>=12){    /* if inch is greater or equal to 12,
                              converting it to feet. */
       d3->inch-=12;
       ++d3->feet;
   }
}
```

Output

```
First distance
Enter feet: 12
Enter inch: 6.8
Second distance
Enter feet: 5
Enter inch: 7.5
Sum of distances = 18'-2.3"
```

Array of Structure

- C language allows creating an array of variables of structure.
- The array of structure is used to store the large number of similar records.
- For example to store the record of 100 employees then array of structure is used.
- The method to define and access the array element of array of structure is similar to other array.
- The syntax to define the array of structure is
  - 'Struct < struct_name > < var_name > < array_name > [< value >];'
- For Example:
  - 'Struct employee e1[100];'

Structures Containing Arrays

- One can define a structure that contains one or more arrays as members.
- The array can be of any C data type (int, char, and so on).
- For example, consider the declaration below.

```
struct data
{
Int x[4];
Char y[10];
};
```

- The above statements define a structure of type data that contains a four-element integer array member named x and a 10-element character array member named y.

Structures and Arrays

- One can then declare a structure named record of type data as follows:
  - struct data record;
- One can access individual elements of arrays that are structure members using a combination of the member operator and array subscripts :
  - record.x[2] = 100;
  - record.y[1] = 'x';

- The character arrays are most frequently used to store strings and the name of an array, without brackets, is a pointer to the array.
- Because this holds true for arrays that are structure members, the expression:
  - record.y
- Is a pointer to the first element of array y[ ] in the structure record.
- Therefore, one could print the contents of y[ ] on-screen using the statement:
  - puts(record.y);

Arrays of Structures

- What one needs is an array of structures of type entry.
- After the structure has been defined, one can declare an array as follows:
  - struct entry list[1000];
- This statement declares an array named list that contains 1,000 elements.
- Each element is a structure of type entry and is identified by subscript like other array element types.
- Each of these structures has three elements, each of which is an array of type char.
- Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.
- Example

```
struct Bookinfo
{
   char[20] bname;
    int pages;
    int price;
}Book[100];
```

- Here Book structure is used to Store the information of one Book.
- In case if we need to store the Information of 100 books then Array of Structure is used.
- b1[0] stores the Information of 1st Book , b1[1] stores the information of 2nd Book and So on We can store the information of 100 books.
- Example Program

```
#include <stdio.h>
struct Bookinfo
{
     char[20] bname;
     int pages;
     int price;
}book[3];
int main(int argc, char *argv[])
{
int i;
for(i=0;i<3;i++)
    {
    printf("\nEnter the Name of Book    : ");
    gets(book[i].bname);
    printf("\nEnter the Number of Pages : ");
    scanf("%d",book[i].pages);
    printf("\nEnter the Price of Book   : ");
    scanf("%f",book[i].price);
    }
printf("\n--------- Book Details ------------ ");
for(i=0;i<3;i++)
    {
    printf("\nName of Book    : %s",book[i].bname);
    printf("\nNumber of Pages : %d",book[i].pages);
    printf("\nPrice of Book   : %f",book[i].price);
    }

return 0;
}
```

Output:

```
Enter the Name of Book    : ABC
Enter the Number of Pages : 100
Enter the Price of Book   : 200
Enter the Name of Book    : EFG
Enter the Number of Pages : 200
Enter the Price of Book   : 300
Enter the Name of Book    : HIJ
Enter the Number of Pages : 300
Enter the Price of Book   : 500

--------- Book Details ------------

Name of Book    : ABC
Number of Pages : 100
Price of Book   : 200
Name of Book    : EFG
Number of Pages : 200
Price of Book   : 300
Name of Book    : HIJ
Number of Pages : 300
Price of Book   : 500
```

Unions

- Unions are similar to structures.
- A union is declared and used in the same ways that a structure.
- The reason for this is simple; all the members of a union occupy the same area of memory.
- Unions are defined and declared in the same fashion as structures.
- In unions all the members share the space which is according to the space requirement of the largest member.
- A union can be initialized on its declaration.
- Because only one member can be used at a time, only one can be initialized.
- To avoid confusion, only the first member of the union can be initialized.
- The following code shows an instance of the shared union being declared and initialised:union shared generic_variable = {'@'};
- The generic variable union was initialized just as the first member of a structure would be initialized.
- Note: The union can hold only one value at a time.

Defining of Union

- A union has to defined, before it can be used.
- The syntax of defining a structure is

```
union <union_name>
 {
<data_type><variable_name>;
<data_type><variable_name>;
   ……..
<data_type><variable_name>;
 };
```

- To define a simple union of a char variable and an integer variable, one would write the following:

```
union shared
{
char c;
int i;
};
```

- This union, shared, can be used to create instances of a union that can hold either a character value c or an integer value i.
- This is an OR condition.
- Unlike a structure that would hold both values, the union can hold only one value at a time.

Union Data Type

- A union is a user defined data type like structure.
- The union groups logically related variables into a single unit.
- The union data type allocates the space equal to space needed to hold the largest data member of union.
- The union allows different types of variable to share same space in memory.
- There is no other difference between structure and union than internal difference.
- The method to declare, use and access the union is same as structure.

```
union Data
{
  int i;
  float f;
  char  str[20];
} data;
```

Accessing Union Members

- Individual union members can be used in the same way that structure members can be used--by using the member operator (.).
- However, there is an important difference in accessing union members.
- Only one union member should be accessed at a time.
- Because a union stores its members on top of each other, it's important to access only one member at a time.
- The following is an example for this:

```
union shared
{
char c;
int i;
};
shared.c='a';
shared.d=1;
```

Difference between Structures & Union

- The memory occupied by structure variable is the sum of sizes of all the members but memory occupied by union variable is equal to space hold by the largest data member of a union.
- In the structure all the members can be accessed at any point of time but in union only one of union member can be accessed at any given time.

```
#include <stdio.h>
union job {        //defining a union
   char name[32];
   float salary;
   int worker_no;
}u;
struct job1 {
   char name[32];
   float salary;
   int worker_no;
}s;
int main(){
   printf("size of union = %d",sizeof(u));
   printf("\nsize of structure = %d", sizeof(s));
   return 0;
}
```

Output

```
size of union = 32
size of structure = 40
```

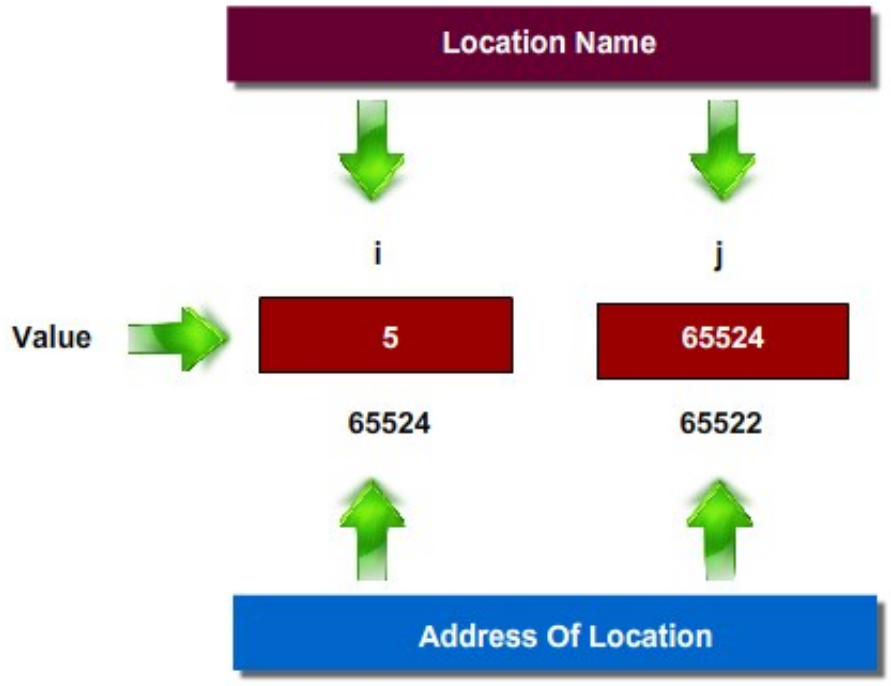
UNIT - 9

Pointers

What are Pointers?

- A pointer is a special variable which holds the address of the variable it has pointed to.

Basic Concept of Pointer



Consider above Diagram

- i is the name given for Particular memory location.
- Consider it's Corresponding address be 65524 and the Value stored in variable ‘i’ is 5
- The address of the variable ‘i’ is stored in another integer variable whose name is ‘j’ and which is having corresponding address 65522
- Thus,one can say that, j = &i; i.e.,j = Address of i.
- Here j is not ordinary variable; It is special variable and called pointer variable as it stores the address of the ordinary variable.
- It can be summarized like

Variable name	Variable value	Variable address
I	5	65524
J	65524	65522

Address operator in C programming

- It is Denoted by '&'
- When used as a prefix to a variable name '&' operator gives the address of that variable.
- Example :
  - &n - Gives address on n.

How Address Operator works ?

```
#include<stdio.h>
void main()
{
        int n = 10;
        printf("\nValue of n is : %d",n);
        printf("\nValue of &n is : %u",&n);
}
```

Output :

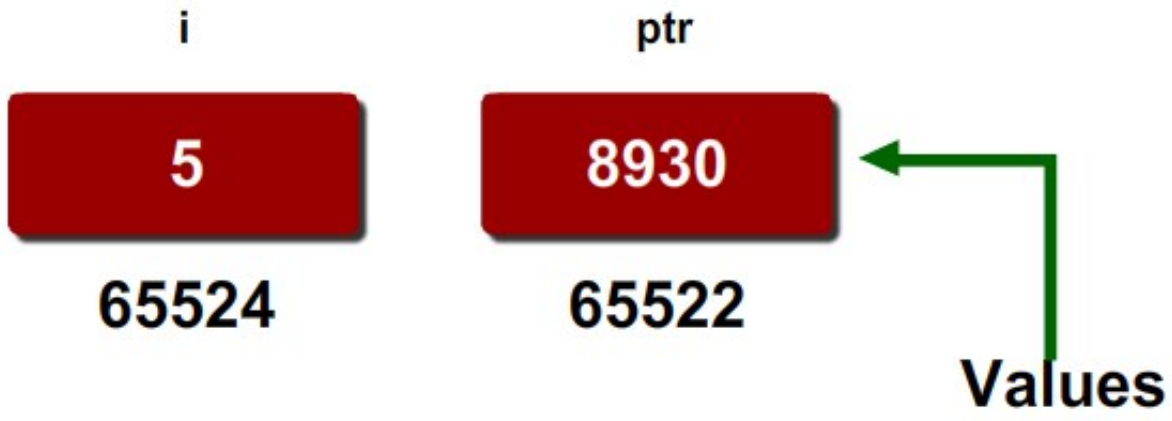
```
Value of  n is : 10
Value of &n is : 1002
```

Visual Understanding

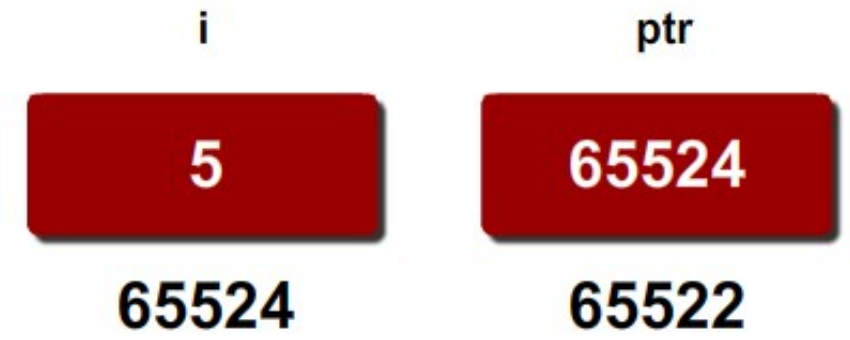
- Consider the following program

```
#include<stdio.h>
int main()
{
        int i = 5;
        int *ptr;
        ptr = &i;
        printf("\nAddress of i    : %u",&i);
        printf("\nValue of ptr is : %u",ptr);
        return(0);
}
```

- After declaration memory map will be like this:
  - int i = 5;
    int *ptr;



- After assigning the address of variable to pointer , i.e after the execution of this statement: ptr = &i;



Invalid Use of Address Operator

- Programmer cannot use Address operator for Accessing Address of Literals.
- Only Variables have Address associated with them.
  - &75
- (a+b) will evaluate addition of Values present in variables.
- Output of (a+b) is nothing but Literal, so one cannot use Address operator.
  - &(a+b)
- Again ‘a’ is Character Literal, so he cannot use Address operator.
  - &(‘a’)

Declaring pointers

- A pointer is a numeric variable and, like all variables, must be declared before it can be used.
- Pointer variable names follow the same rules as other variables and must be unique.
- This chapter uses the convention that a pointer to the variable name is called p_name.
- This isn't necessary, however; one can name pointers to anything within C's naming rules.
- A pointer declaration takes the following form:
  - typename *ptrname;
- Typename is any of C's variable types and indicates the type of the variable that the pointer points to.
- The asterisk (*) is the indirection operator, and it indicates that ptrname is a pointer to type typename and not a variable of type typename.
- Pointers can be declared along with non-pointer variables.
- char *ch1, *ch2; /* ch1 and ch2 both are pointers to type char */

Syntax for Pointer Declaration in C

- data_type *< pointer_name >;

Explanation

- data_type
- Type of variable that the pointer points to
- OR data type whose address is stored in pointer_name.
  - Asterisk(*).
- Asterisk is called as Indirection Operator.
- It is also called as Value at address Operator.
- It Indicates Variable declared is of Pointer type.
  - pointer_name.
- Must be any Valid C identifier.
- Must follow all Rules of Variable name declaration.

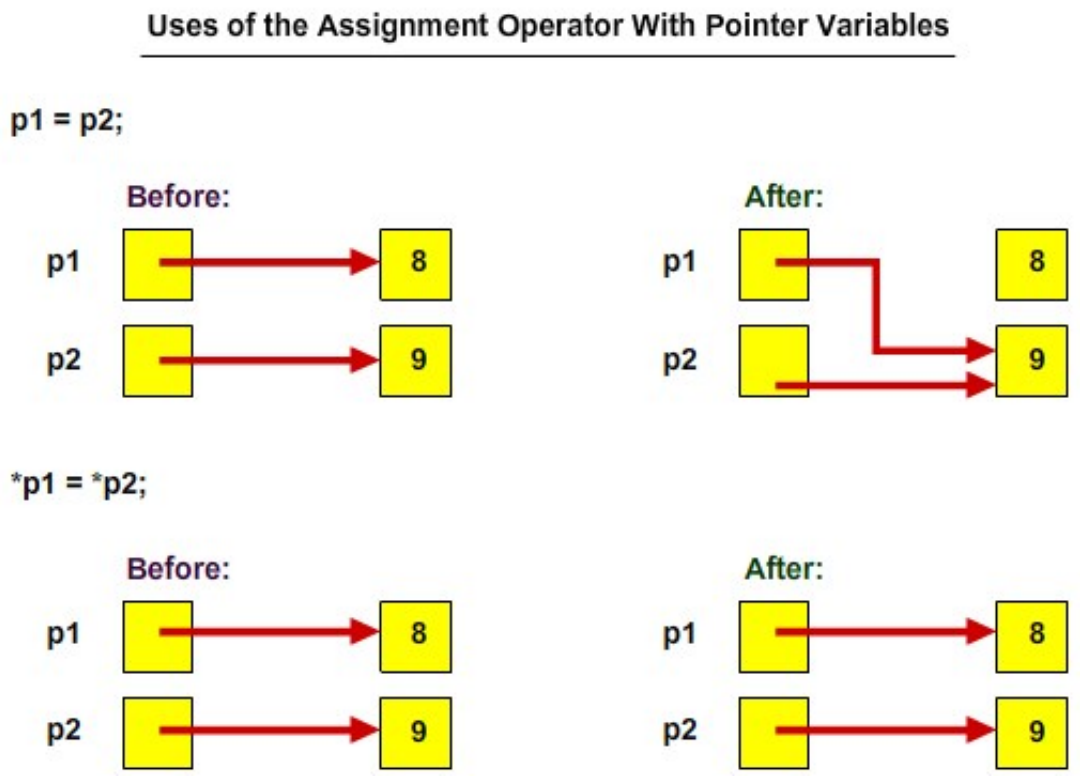
Ways of Declaring Pointer Variable

- * can appears anywhere between Pointer_name and Data Type.

```
int *p;
int *     p;
int   * p;
```

Pointer Assignments

- To assign an address to a pointer we'll need a new operator, the "address of" operator.
- Once assigned, the pointer will contain the "address of" the assigned variable not it's value.
- Code Example:
  - int *xp; // declares xp as a pointer to an integer
  - xp = &x; // xp receives the address of 'x'
- Pointer variables can be "assigned":
  - int *p1, *p2;
  - p2 = p1;
  - Assigns one pointer to another.
  - "Make p2 point to where p1 points".
- Do not confuse with:
  - *p1 = *p2;
  - Assigns "value pointed to" by p1, to "value pointed to" by p2.



Example program: Pointer Assignments

```
#include <stdio.h>

int main ()
{
   /* x is an integer variable. */
   int x = 42;
   /* x_ptr is a pointer to an integer variable. */
   int * x_ptr = & x;
   printf ("x = %d\n", x);
   printf ("x_ptr = %p\n", x_ptr);
   return 0;
}
```

Initialization

How to Initialize Pointer in C Programming?

- pointer = &variable;
- Above is the syntax for initializing pointer variable in C.

Initialization of Pointer can be done using following 4 Steps

- Declare a Pointer Variable and Note down the Data Type.
- Declare another Variable with Same Data Type as that of Pointer Variable.
- Initialize Ordinary Variable and assign some value to it.
- Now initialize pointer by assigning the address of ordinary variable to pointer variable.
- The following example will clearly explain the initialization of Pointer Variable.

```
#include<stdio.h>
int main()
{
        int a;
        int *ptr;     // pointer declaration
        a = 10;       // Assign some value
        ptr = &a;     //pointer initialization
        return(0);
}
```

Explanation of Above Program

- Pointer should not be used before initialization.
- “ptr” is pointer variable used to store the address of the variable.
- Stores address of the variable ‘a’.
- Now “ptr” will contain the address of the variable “a”.
- Note :Pointers are always initialized before using it in the program.

Example : Initializing Integer Pointer

```
#include<stdio.h>
int main()
{
        int a = 10;
        int *ptr;

        ptr = &a;
        printf("\nValue of ptr : %u",ptr);

        return(0);
}
```

Output :

```
Value of ptr : 4001
```

- Like regular variables, uninitialized pointers can be used, but the results are unpredictable and potentially disastrous.
- Until a pointer holds the address of a variable, it isn't useful.
- The address doesn't get stored in the pointer .
- The program must put it there by using the address-of operator, the ampersand (&).
- When placed before the name of a variable, the address-of operator returns the address of the variable.
- Therefore, one has to initialize a pointer with a statement of the form:pointer = &variable;
- The program statement to initialize the variable p_rate to point at the variable rate would be:p_rate = &rate; /* assign the address of rate to p_rate */
- Before the initialization, p_rate didn't point to anything in particular. After the initialization, p_rate is a pointer to rate.

Pointers and Variable Types

- For the more common PC operating systems, an int takes two bytes; a float takes four bytes, and so on.
- Each individual byte of memory has its own address, so a multibyte variable actually occupies several addresses.
- How, then, do pointers handle the addresses of multibyte variables?
- Here's how it works: The address of a variable is actually the address of the first (lowest) byte it occupies.
- This can be illustrated with an example that declares and initializes three variables:

```
int vint = 123456;
char vchar = 90;
float vfloat = 1234.15;
```

- These variables are stored in memory.
- In this, the int variable occupies two bytes, the char variable occupies one byte, and the float variable occupies four bytes.
- Different types of numeric variables occupy different amounts of storage space in memory.

```
int *p_vint;
char *p_vchar;
float *p_vfloat;
```

Pointer Arithmetic

- See table and observe -Have declared some of the variables and also assumed some address for declared variables.

Data Type	Initial Address	Operation	Address After Operation	Required Bytes
int	4000	++	4002	2
int	4000	- -	3998	2
char	4000	++	4001	1
char	4000	- -	3999	1
float	4000	++	4004	4
float	4000	- -	3996	4
long	4000	++	4004	4
long	4000	- -	3996	4

- One can see address of a variable after performing arithmetic operations.

Expression	Result
Address + Number	Address
Address - Number	Address
Address - Address	Number
Address + Address	Illegal

- It clearly shows that one can add or subtract address and integer number to get valid address.
- A programmer can even subtract two addresses but he cannot add two addresses.

Arithmetic Pointers (Incrementing)

- When one increments a pointer, he is increasing its value.
- For example, when one increments a pointer by 1, pointer arithmetic automatically increases the pointer's value so that it points to the next array element.
- In other words, C knows the data type that the pointer points to (from the pointer declaration) and increases the address stored in the pointer by the size of the data type.

- Suppose that ptr_to_int is a pointer variable to some element of an int array.
- If execute the statement ptr_to_int++; the value of ptr_to_int is increased by the size of type int (usually 2 bytes), and ptr_to_int now points to the next array element.
- Likewise, if ptr_to_float points to an element of a type float array, the statement: ptr_to_float++;
- Increases the value of ptr_to_float by the size of type float (usually 4 bytes).
- The same holds true for increments greater than 1.
- If the value n is added to a pointer, C increments the pointer by n array elements of the associated data type.
- Therefore: ptr_to_int += 4;
- Increases the value stored in ptr_to_int by 8 (assuming that an integer is 2 bytes), so it points four array elements ahead. Likewise: ptr_to_float += 10;
- Increases the value stored in ptr_to_float by 40 (assuming that a float is 4 bytes), so it points 10 array elements ahead.

Arithmetic Pointers(incrementing) - Program

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = var;
  for ( i = 0; i < MAX; i++)
  {
    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );
    /* move to the next location */
    ptr++;
  }
  return 0;
}
```

Arithmetic Pointers (decrementing)

- The same concepts that apply to incrementing pointers hold true for decrementing pointers.
- Decrementing a pointer is actually a special case of incrementing by adding a negative value.
- If one decrements a pointer with the -- or - = operators, pointer arithmetic automatically adjusts

for the size of the array elements.

Arithmetic Pointers (decrementing)

- Suppose that ptr_to_int is a pointer variable to some element of an int array.
- If this statement is executed: ptr_to_int--; the value of ptr_to_int is decreased by the size of type int (usually 2 bytes), and ptr_to_int now points to the next array element.
- Likewise, if ptr_to_float points to an element of a type float array, the statement: ptr_to_float--; decrease the value of ptr_to_float by the size of type float (usually 4 bytes).

Arithmetic Pointers (decrementing) – program

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
  int  var[] = {10, 100, 200};
  int  i, *ptr;

  /* let us have array address in pointer */
  ptr = &var[MAX-1];
  for ( i = MAX; i > 0; i--)
  {

    printf("Address of var[%d] = %x\n", i, ptr );
    printf("Value of var[%d] = %d\n", i, *ptr );

    /* move to the previous location */
    ptr--;
  }
  return 0;
}
```

Other Pointer Manipulations

- The only other pointer arithmetic operation is called differencing, which refers to subtracting two pointers.
- If one has two pointers to different elements of the same array, one can subtract them and find out how far apart they are.
- Again, pointer arithmetic automatically scales the answer so that it refers to array elements.
- Thus, if ptr1 and ptr2 point to elements of an array (of any type), the following expression tells how far apart the elements are: ptr1 – ptr2.
- Pointer comparisons are valid only between pointers that point to the same array.

- Under these circumstances, the relational operators ==, ! =, >, <, >=, and <= work properly.
- Lower array elements (that is, those having a lower subscript) always have a lower address than higher array elements.
- Thus, if ptr1 and ptr2 point to elements of the same array, the comparison:ptr1 < ptr2.
- Is true if ptr1 points to an earlier member of the array than ptr2 does.

Pointers and Functions

- When the program is executed, the code for each function is loaded into memory starting at a specific address.
- A pointer to a function holds the starting address of a function-- its entry point.
- The code for each function is loaded into memory starting at a specific address.
- Like other pointers, one must declare a pointer to a function.
- The general form of the declaration is as follows:
  - type (*ptr_to_func)(parameter_list);

What is function Pointer?

- Function pointer: A pointer which keeps address of a function is known as function pointer.

Return Type : None Parameter : None	void*(*ptr)();	ptr = &display;	(*ptr)();
Return Type : Integer Parameter : None	int*(*ptr)();	ptr = &display;	int result; result = (*ptr)();
Return Type : Float Parameter : None	float*(*ptr)();	ptr = &display;	float result; result = (*ptr)();
Return Type : Char Parameter : None	char*(*ptr)();	ptr = &display;	char result; result = (*ptr)();

- Three steps to use pointer to call function:
  - Declare Pointer which is capable of storing address of function.
  - Initialize Pointer Variable.
  - Call function using Pointer Variable.

Step 1 : Declaring Pointer

- void *(*ptr)();
  - Declare a double pointer.
  - Write () symbol after “Double Pointer“.
  - void represents that, function is not returning any value.
  - () represents that, function is not taking any parameter.

Step 2 : Initializing Pointer

- ptr = &display;
  - This statement will store address of function in pointer variable.

Step 3 : Calling a function

- (*ptr)();
  - using (*ptr)(),one can call function display();

```
(*ptr)() = (*ptr)();
         = (*&display)();
         = (display)();
         = display();
```

- Example: Function having two Pointer Parameters and return type as Pointer.

```
#include<stdio.h>
char * getName(int *,float *);
int main()
{
char *name;
int  num = 100;
float marks = 99.12;
char *(*ptr)(int*,float *);
ptr=&getName;
name = (*ptr)(&num,&marks);
printf("Name : %s",name);
return 0;
}
//-----------------------------------
char *getName(int *ivar,float *fvar)
{
char *str="Pointer Demo";
str = str + (*ivar) + (int)(*fvar);
return(str);
}
```

- This statement declares ptr_to_func as a pointer to a function that returns type and passes the parameters in parameter list.

- Here are some more concrete examples:

```
int(*func1)(int x);

void(*func2)(double y, double z);

char(*func3)(char *p[]);

void(*func4)();
```

- The first line declares func1 as a pointer to a function that takes one type int argument and returns a type int.
- The second line declares func2 as a pointer to a function that takes two type double arguments and has a void return type (no return value).
- The third line declares func3 as a pointer to a function that takes an array of pointers to type char as its argument and returns to type char.
- The final line declares func4 as a pointer to a function that doesn't take any arguments and has a void return type.

Initializing and Using a Pointer to a Function

- A pointer to a function must not only be declared, but also initialized to pointer something.
- The "something" is, of course, a function.
- There's nothing special about a function that gets pointed to.
- The only requirement is that its return type and parameter list match the return type and parameter list of the pointer declaration.
- For example, the following code declares and defines a function and a pointer to that function:

```
float square(float x); /* The function prototype. */

float(*p)(float x); /* The pointer declaration */

float square(float x) /* The function definition. */

{

Return x*x;

}
```

- Because the function square() and the pointer p have the same parameter and return types, one can initialize p to point to square as follows: p = square;

- Then a programmer can call the function using the pointer as follows: answer = p(x);

The Array name - as Pointers

- An array name without brackets is a pointer to the array's first element.
- Thus, if a programmer has declared an array data[ ], data is the address of the first array element.
- For example, the following code initializes the pointer variable p_array with the address of the first element of array[ ].

```
int array[100], *p_array;

/* additional code goes here*/

p_array = array;
```

- p_array is a pointer variable, it can be modified to point elsewhere.
- Unlike array, p_array isn't locked into pointing at the first element of array[ ].
- Example, it could be pointed at other elements of array[ ].

Pointer Pointing to the Array of Elements

- Array and Pointer are backbones of the C Programming language.
- Pointer and array , both are closely related.
- Array name without subscript points to first element in an array.

Example : One Dimensional Array

- int a[10];
  - Here a , a[0] both are identical.

Example : Two Dimensional Array

- int a[10][10];
  - Here a , a[0][0] both are identical.
  - Let x is an array , i is subscript variable.

```
x [ i ]  = * ( x + i )
         = * ( i + x )
         =   i [ x ]
```

Proof of the above Formula

- For first element of array i = 0

$$x[0] = *(x + 0)$$

$$= *x$$

- The first element of array is equivalent to *x .
- Thus *(x+i) will gives us value of ith element.

Arrays and Pointers

- An array name without brackets is a pointer to the array′s first element.
- Therefore, one can access the first array element using the indirection operator.
- If array[ ] is a declared array, the expression *array is the array′s first element, *(array + 1) is the array′s second element, and so on.
- This illustrates the equivalence of array subscript notation and array pointer notation.
- If one can generalize for the entire array, the following relationships hold true:

```
*(array)==array[0]
*(array+1)==array[1]
*(array+2)==array[2]
...
*(array+n)==array[n]
```

Array Subscript Notation and Pointers

```
#include <stdio.h>
int main ()
{
  /* an array with 5 elements */
  double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
  double *p;
  int i;

  p = balance;

  /* output each array element's value */
  printf( "Array values using pointer\n");
  for ( i = 0; i < 5; i++ )
  {
     printf("*(p + %d) : %f\n",  i, *(p + i) );
  }

  printf( "Array values using balance as address\n");
  for ( i = 0; i < 5; i++ )
  {
     printf("*(balance + %d) : %f\n",  i, *(balance + i) );
  }

  return 0;
}
```

Strings and Pointers

- To allocate and initialize a string is to declare an array of type char as follows:
  - char message[ ] = "This is the message.";
  - One could accomplish the same thing by declaring a pointer to type char:
  - char *message = "This is the message.";

Strings and Pointers – program

```
#include<stdio.h>

int main()
{
int i;

char *arr[4] = {"C","C++","Java","VBA"};
char *(*ptr)[4] = &arr;

for(i=0;i<4;i++)
   printf("String %d : %s\n",i+1,(*ptr)[i]);

return 0;
}
```

Array of Pointers to Type char

- The following statement declares an array of 10 pointers to type char:
  - char *message[10];
- Each element of the array message[ ] is an individual pointer to type char.
- One can combine the declaration with initialisation and allocation of storage space for the strings:
  - char *message[10] = { "one", "two", "three" };
- This declaration does the following:
  - It allocates a 10-element array named message; each element of message is a pointer to type char.
  - It allocates space somewhere in memory and stores the three initialization strings, each with a terminating null character.
  - It initializes message[0] to point to the first character of the string "one", message[1] to point to the first character of the string "two", and message[2] to point to the first character of the string "three".

Array of Pointers

- Step 1 : Array of Function
  - int(*arr[3])();
    - Above statement tell that arr is an array of size 3.
    - Array stores address of functions.

- Array stores address of function having integer as return type and does not takes any parameter.

- Step 2 : Declaring Array of function Pointer
  - int(*(*ptr)[3])();
    - It is array of function pointer which points to “array of function“.
- Step 3 : Store function names inside function array
  - arr[0] = display;
  - arr[1] = getch;
- Step 4 : Store address of function Array to Function Pointer
  - ptr = &arr;
- Step 5 : Calling Function
  - Following syntax is used to call display function: (**ptr)();
    - This syntax is used to call getchfunction:v(*(*ptr+1))();

Pointers to Structures

- A structure is a collection of variables within one variable.
- Structures can be termed as a variable that aggregates variables of different or similar types.
- The correlation of structures and pointers is very strong.
- A structure provides a very intuitive way to model user-defined entities (records, packet formats, image headers, etc.).

Defining Structures

```
struct name
{

member1;
member2;
 . .

};

-------- Inside function -------

struct name *ptr;
```

Example Program: Pointers to Structures

```
struct Book
{
char name[10];
int price;
}
int main()
{
struct Book a;      // Single structure variable
struct Book* ptr;  //Pointer of structure type
Ptr = &a;
struct Book b[10];
struct Book* p;
p=&b;
}
```

Dynamic Memory Allocation

- In general, there are two types of storages available for variables.
- They are
  - (i) Stack and
  - (ii) Heap memory storages.
- The stack memory is permanent storage area where the ordinary variables can be stored.
- The heap memory is the free memory area available in the main memory of the computer which will be used for dynamic memory allocation.
- In C, there exists a set of built-in functions which are used to allocate memory dynamically to the derived data types like arrays, structures and unions.
- Some of the important functions are:
  - (i) malloc ()
  - (ii) calloc () and
  - (iii) free ()

malloc ()

- The general syntax of the malloc () function is
  - pointer variable = (data-type *) malloc (no. of bytes to be allocated);

Example

- int *x, n;
  x=(int *) malloc(n * sizeof (int));
- Here, x is the pointer variable of int data type and n is an int type variable which can hold the value of the size of the array to be read-in.
- The malloc() function allocates a single contiguous memory of n locations of int type for the pointer variable x, such that x can become like an array of n int values, and the memory location address of the first element of the array is stored in x.
- This will provide the understanding of the connection between the pointers and arrays.
- The malloc() function does not initialize the allocated memory by default.
- So, one has to perform initialization separately before using them.
- For the above example, it can be done simply as we do initialization for any array and is given below.

- for (i = 0; i < n; i++)
  x[i] = 0;

- In general, malloc () function is used to allocate memory for basic data types like int,char, float, and double.
- In order to allocate memory dynamically for derived data types like structures and unions calloc () function is used.

calloc()

- The general syntax of the calloc () function is
  - pointer variable = (data-type *) calloc (no. of bytes to be allocated);

Example

- int *x, n;
  x = (int *) calloc(n, sizeof(int));
- The above example is similar to the malloc () function, but with the following changes. Here there are two arguments that are used to specify the number of bytes to be allocated.
- The calloc function allocates a number of blocks of memory in contiguous form and not as a single contiguous block of memory allocated by malloc() function.
- Since the calloc () function allocates memory as number of blocks, even in the case of single contiguous block of memory is not available.
- The other important aspect is that the calloc () function initializes the allocated memory, by default.
- So, one need not initialize the allocated memory as in the case of malloc() function.
- This calloc () function is used to allocate memory for derived data types like arrays, structures and unions.

Differences between malloc () and calloc () functions

- The main differences between the two dynamic memory allocation functions malloc ( ) and calloc () available in C are given in Table:

malloc ()	calloc ()
It allocates single contiguous block of memory only.	It allocates number of blocks of memory in contiguous form.
Memory allocation is not initialized, by default.	Memory allocation is initialized, by default.
It is used to allocate memory for basic data types (like int, char, float and double).	It is used to allocate memory for derived data types (like arrays, structures and unions).

free()

- Dynamically allocated memory with either calloc() or malloc() does not get return on its own.
- The programmer must use free() explicitly to release space.

Syntax of free()

- free(ptr);
- This statement causes the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

- Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
if(ptr==NULL)
{
printf("Error! memory not allocated.");
exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
scanf("%d",ptr+i);
sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

- Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main(){ int n,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)calloc(n,sizeof(int));
if(ptr==NULL)
{
printf("Error! memory not allocated.");
exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;++i)
{
scanf("%d",ptr+i);
sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}
```

UNIT - 10

Self Referential Structures and Linked Lists

Self-referential Structures

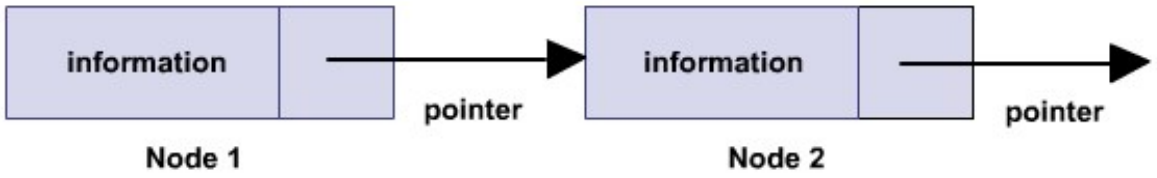
- A structure can have members which point to a structure variable of the same type.
- These types of structures are called self-referential structures and are widely used in dynamic data structures like trees, linked list, etc.
- The following is a definition of a self-referential structure.

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

- Here, next is a pointer to a struct node variable.
- It should be remembered that a pointer to a structure is similar to a pointer to any other variable.
- A self-referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind.
- A chain of such structures can thus be expressed as follows.

```
struct name
{
   member 1;
   member 2;
   . . .
   struct name *pointer;
};
```

- The above illustrated structure prototype describes one node that comprises of two logical segments.
- One of them stores data/information and the other one is a pointer indicating where the next component can be found.
- Several such inter-connected nodes create a chain of structures.
- The following figure depicts the composition of such a node.
- The figure is a simplified illustration of nodes that collectively form a chain of structures or linked list.

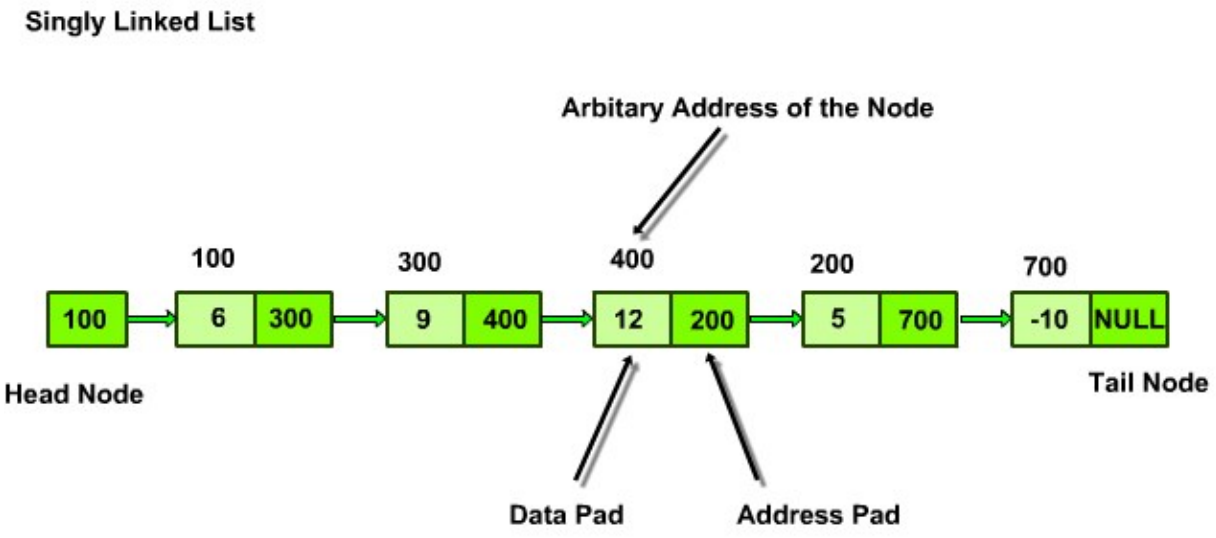


- Such self-referential structures are very useful in applications that involve linked data structures, such as lists and trees.
- Unlike a static data structure such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self-referential structure can dynamically be expanded or contracted.
- Operations like insertion or deletion of nodes in a self- referential structure involve simple and straight forward alteration of pointers.

Linked Lists

- A linked list is a useful method of data storage that can easily be implemented in C.
- There are several kinds of linked lists, including single-linked lists, double-linked lists, and binary trees.
- Each type is suited for certain types of data storage.
- The one thing that these lists have in common is that the links between data items are defined by information that is contained in the items themselves, in the form of pointers.
- This is distinctly different from arrays, in which the links between data items result from the layout and storage of the array.

Creation of a Singly Connected Linked List



Basics of Linked Lists

- Each data item in a linked list is contained in a structure.
- The structure contains the data elements needed to hold the data being stored; these depend on the needs of the specific program.
- In addition, there is one more data element--a pointer.
- This pointer produces the links in a linked list. Here's a simple example:

```
struct person
{
char name[20];
struct person *next;
};
```

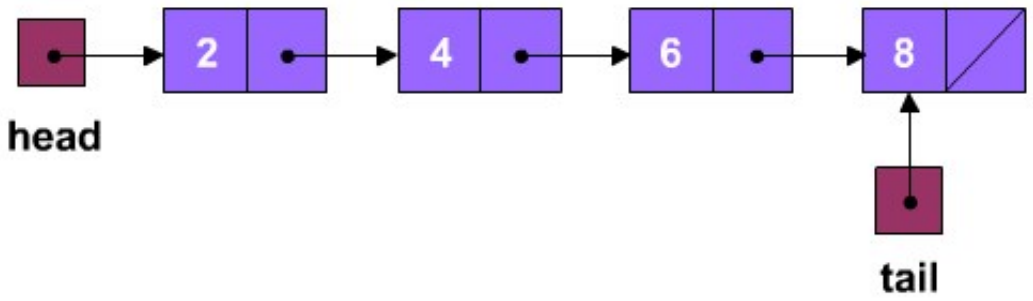
- This code defines a structure named person.

- For the data, person contains only a 20-element array of characters.
- One generally wouldn't use a linked list for such simple data, but this will serve as an example.
- The person structure also contains a pointer to type person--in other words, a pointer to another structure of the same type.
- This means that each structure of type person can not only contain a chunk of data, but also can point to another person structure.

Head Pointer

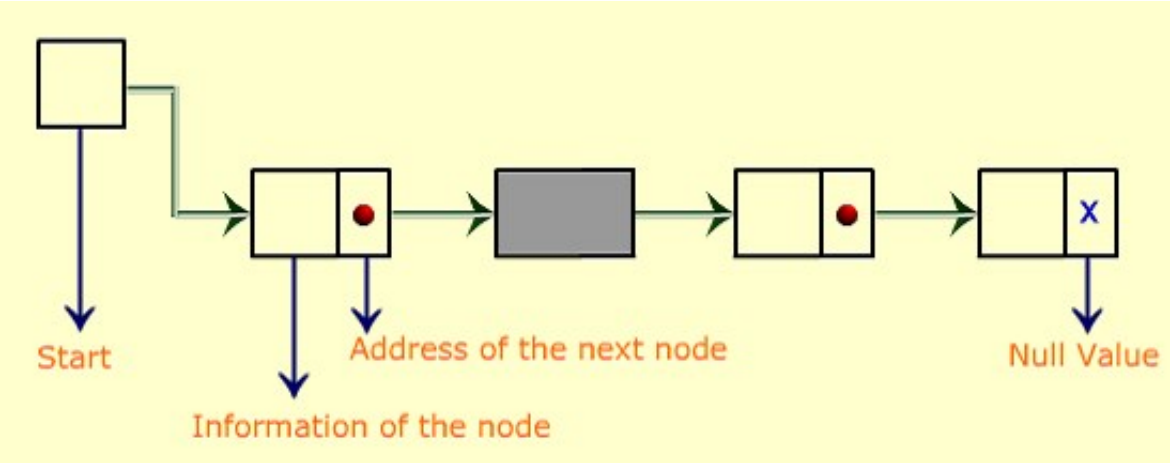
- This is identified by a special pointer (not a structure) called the head pointer.
- The head pointer always points to the first element in the linked list.
- The first element contains a pointer to the second element; the second element contains a pointer to the third, and so on until one encounters an element whose pointer is NULL.
- If the entire list is empty (contains no links), the head pointer is set to NULL.
- NOTE: The head pointer is a pointer to the first element in a linked list.
- The head pointer is sometimes referred to as the first element pointer or top pointer.

Self-Referential Structures and Linked Lists - Head and Tail



Traversing Linked List

- A linked list is a data structure that is used to model such a dynamic list of data items, so the study of the linked lists as one of the data structures is important.
- Linked list is essentially an ordered list of some data in which one can traverse it (examine it) in only one direction.
- Linked list is made up of nodes that consist of data section that holds the actual data and a link section, which contains information about the next node in the list.
- Usually, that link section contains a pointer to the next node, or in array implementation of the list, an index of the next node.
- Usually Linked List has the following functions:
  - Add – adds an element at the end of the list.
  - Add (position) – adds an element at a certain position.
  - Remove (position) – removes a certain element.
  - Traverse – traverses the list and possibly display its content on the screen.
  - IsEmpty – function that checks if the list is empty or not and possibly others such as getSize, etc.
- It is important to check the position parameter when Add (position) and Remove (position) functions are called.
- Suppose the list has 5 elements in total – 0 to 4.
- It would be legal to call Add or Remove with position parameter in the range of 0 to 4 and also it would be legal to call Add with the position parameter value of 5, which would essentially add a new Node at the very end of the list (after the 4th element).
- Thus in the above linked list of 4 elements Node 0 has a pointer to Node 1, Node 1 to Node 2, Node 2 to Node 3.
- Also need to keep a pointer to the first node in the list, First Node (which is Node 0 in the above example) to be able to access the list at all.
- Clearly, next is used to create a link to the next node in the list, hence the name linked list.
- Linked list can be represented graphically as follows:



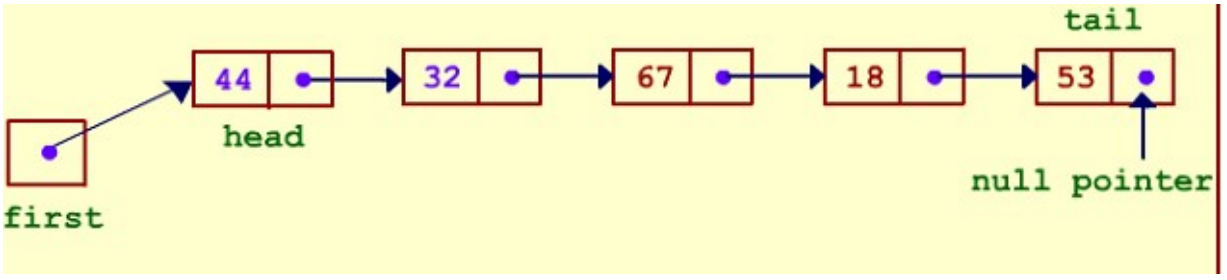
- Steps for Traversing the linked list:
  - Step1: Start at the beginning – First Node.
  - Step2: Access node's data section.
  - Step3: Print out the data or does some other work per application specification.
  - Step4: Access the next node by using the link provided in the next section of the current node.
- Repeat step 2.
- Do steps 2 and 3 as long as the last node is not reached.
- Last node will point to nothing in its next section.
- Nothing is represented variously depending on the actual implementation.

Insertion into a Linked List

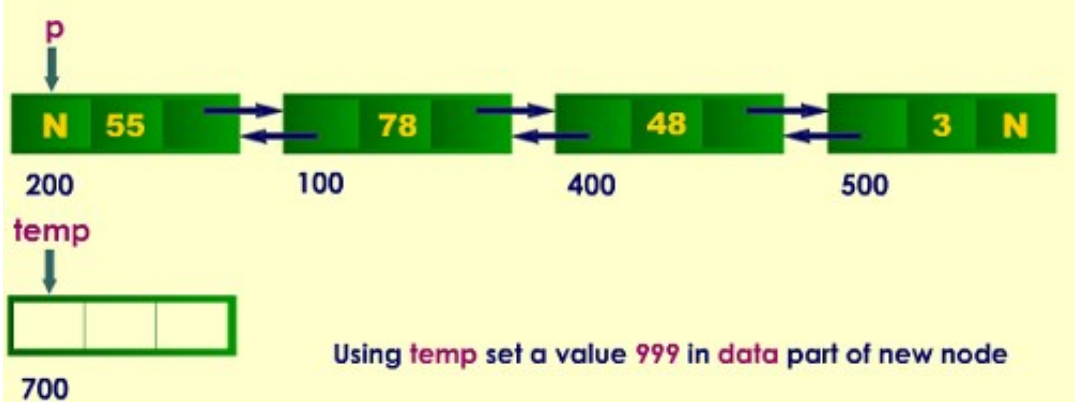
- Adding an Element to the Beginning of a List
  - If the head pointer is NULL, the list is empty, and the new element will be its only member.
  - If the head pointer is not NULL, the list already contains one or more elements.
  - In either case, however, the procedure for adding a new element to the start of the list is the same:
- Create an instance of the structure, allocating memory space using malloc().
- Set the next pointer of the new element to the current value of the head pointer.
- This will be NULL if the list is empty, or the address of the current first element otherwise.
- Make the head pointer point to the new element.
- Here is the code to perform this task:

```
new=(person*)malloc(sizeof(struct person));

new→next=head;

head=new;
```

- Adding an Element to the Beginning of a List: Example



- Adding an Element to the Beginning of a doubly Linked List: Example

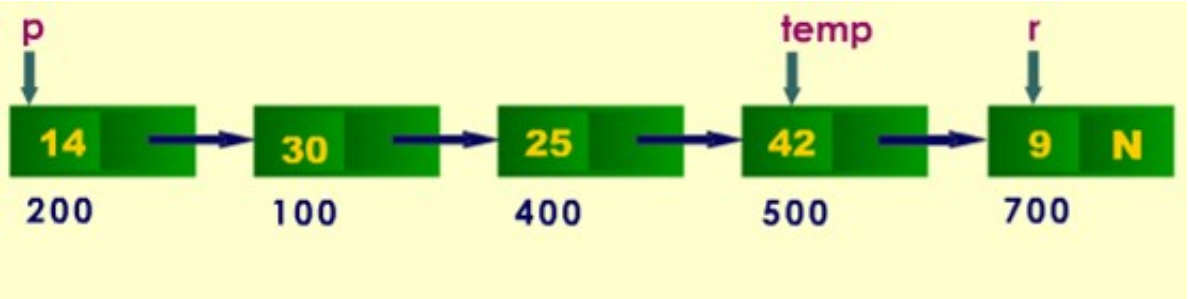


WARNING:

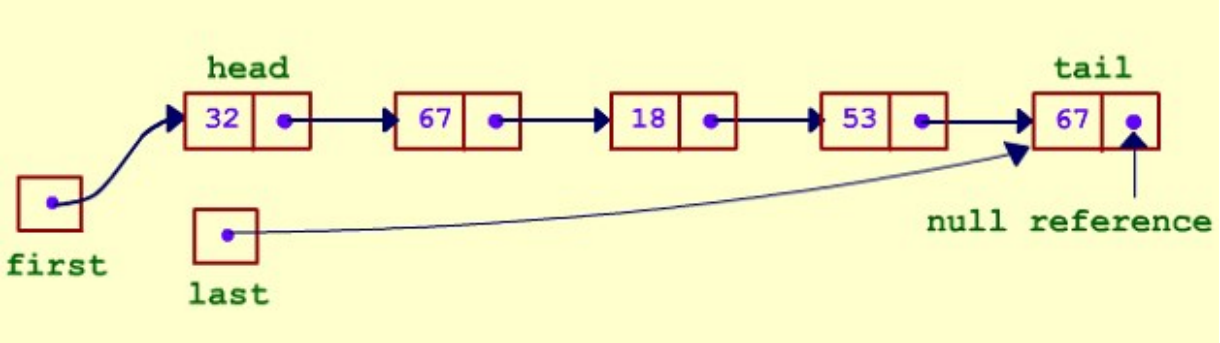
- It is important to switch the pointers in the correct order.
- If one re-assigns the head pointer first, one will lose the list.
- The malloc() is used to allocate the memory for the new element.
- As each new element is added, only the memory needed for it is allocated.
- The calloc() function could also be used.
- One should be aware of the differences between these two functions.
- The main difference is that calloc() will initialize the new element; malloc() will not.
- Adding an Element to the End of the List
  - To add an element to the end of a linked list, one needs to start at the head pointer and go through the list until one finds the last element.
  - Create an instance of the structure, allocating memory space using malloc().
  - Set the next pointer in the last element to point to the new element (whose address is returned by malloc()).
  - Set the next pointer in the new element to NULL to signal that it is the last item in the list.
- Here is the code:

```
person *current;
...
current=head;

while(current→next!=NULL)

current=current→next;

new=(person*)malloc(sizeof(struct person));

current→next=new;

new→next=NULL;
```

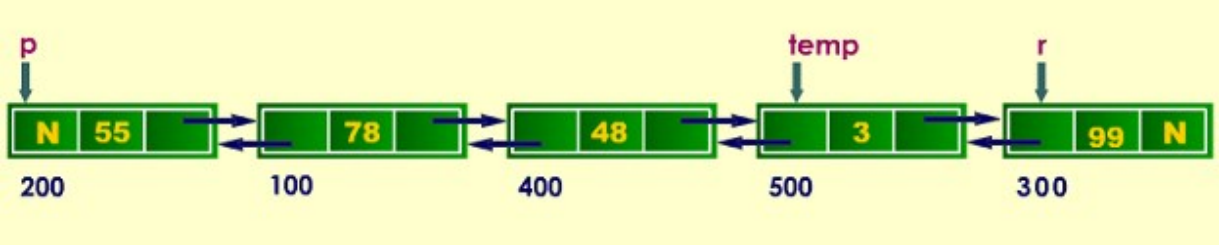
- Example:



- Adding an Element to the End of the singly linked list: Example:



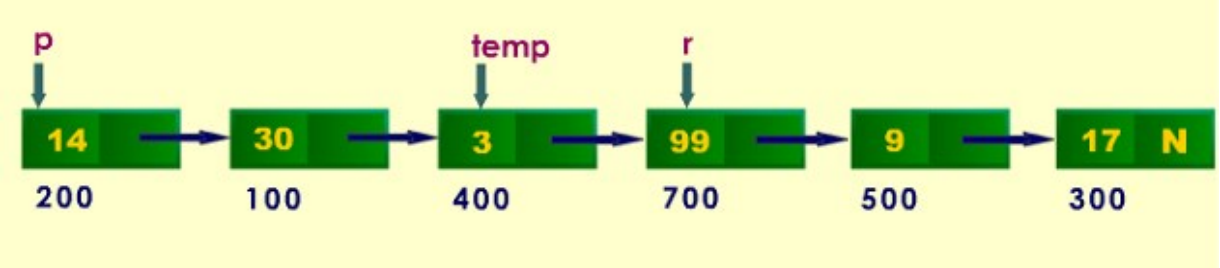
- Adding an Element to the End of a doubly Linked ListExample:



- Adding an Element to the Middle of the List
  - When one is working with a linked list, most of the time one will be adding elements somewhere in the middle of the list.
  - Exactly where the new element is placed depends on how one keeps the list--for example, if it is sorted on one or more data elements.
  - This process, then, requires that one first locates the position in the list where the new element will go, and then add it.
  - In the list, locate the existing element that the new element will be placed after. Let's call this the marker element.
  - Create an instance of the structure, allocating memory space using malloc().
  - Set the next pointer of the new element to point to the element that the marker element used to point to.
  - Set the next pointer of the marker element to point to the new element . (whose address is returned by malloc()).
  - Here's how the code might look:

```
person *marker;

/*Code here to set marker to point to the desired list location. */
...
new=(LINK)malloc(sizeof(PERSON));

new→next = marker→next;

marker→next = new;
```

- Example:

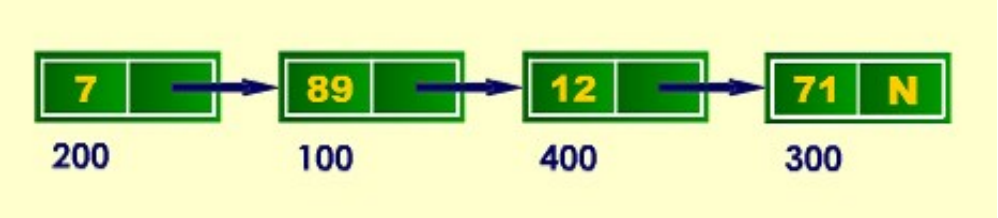


Deletion from a Linked List

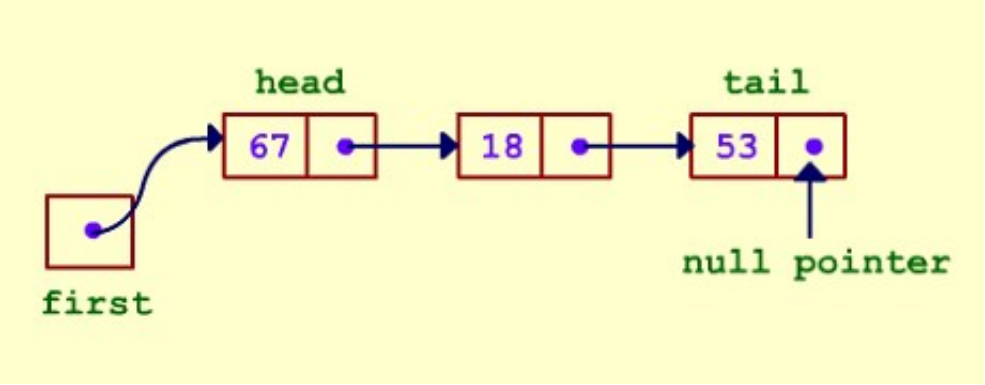
- Deleting an element from a linked list is a simple matter of manipulating pointers.
- The exact process depends on where in the list the element is located:
  - To delete the first element, set the head pointer to point to the second element in the list.
  - To delete the last element, set the next pointer of the next-to-last element to NULL.
  - To delete any other element, set the next pointer of the element before the one being deleted to point to the element after the one being deleted.
- Here's the code to delete the first element in a linked list: head = head->next;

Example

- Deletion of a node from a linked list



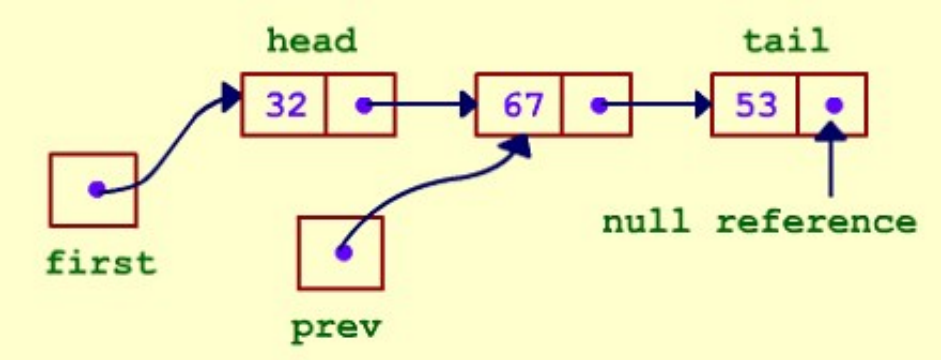
- Deleting an element from the list



- The following are the codes to delete the last element and the element in the middle of the list:

```
person *current1, *current2;

current1=head;

current2=current1→next;

while(current2→next!=NULL)
{
current1=current2;
current2=current1→next;
}

current1→next=null;

if(head==current1);

head=null;
```

- Deleting from the middle of a list



- The following code deletes an element from within the list:

```
person *current1, *current2;

/* Code goes here to have current1 point to the
    Element just before the one to be deleted*/

current2=current1→next

current1→next=current2→next;
```

- After any of these procedures, the deleted element still exists in memory, but it is removed from the list because there is no pointer in the list pointing to it.
- This is accomplished with the free() function.

UNIT - 11

File Processing

Concept of Files - What are files?

- Many real-world problems handle large volume of data and in such situations external storage devices like the floppy disk and the hard disks are used.
- Data is stored in these devices using the concept of files.
- A file is a collection of related data stored on a particular area of the disk.

Filenames

- Every disk file has a name, and one must use filenames when dealing with disk files.
- Filenames are stored as strings, just like other text data.
- The rules as to what is acceptable for filenames and what is not, differ from one operating system to another.

File Opening in Various Modes and Closing of a File

Opening a File

- The process of creating a stream linked to a disk file is called opening the file.
- When one opens a file, it becomes available for reading (meaning that data is input from the file to the program), writing (meaning that data from the program is saved in the file), or both.
- After working with the file, close the file.
- To open a file, use the fopen() library function.

```
Void main()

FILE *fp;

fp=fopen("test","a");

fprintf(fp,"%d %s %d\n",1,"asdfasdf",2);

fclose(fp);

}
```

- The prototype of fopen() is located in STDIO.H and it reads as follows:
- This prototype tells the programmer that fopen() returns a pointer to type FILE, which is a structure declared in STDIO.H.
- The members of the FILE structure are used by the program in the various file access operations; there is no need to be concerned about them.

```
#include <stdio.h>
int main()
{
FILE *file;
file = fopen("abc.c", "r");
if(file==NULL)
{
   printf("Error: can't open file.\n");
   return 1;
   }
   else
   {
      printf("File opened. Now closing it...\n");
      fclose(file);
      return 0;
   }
}
```

- However, for each file that has to be open, one must declare a pointer to type FILE.
- The fopen(), that function creates an instance of the FILE structure and returns a pointer to that structure.
- A programmer uses this pointer in all subsequent operations on the file.
- If fopen() fails, it returns NULL.
- The argument filename is the name of the file to be opened.
- As noted earlier, filename can—and should--contain a path specification.
- The filename argument can be a literal string enclosed in double quotation marks or a pointer to a string variable.
- The argument mode specifies the mode in which to open the file.
- In this context, mode controls whether the file is binary or text and whether it is for reading, writing, or both.
- The default file mode is text.
- Values of mode for the fopen() function.
  - FILE *ptrfile.

    ptrfile = fopen("filename","mode").

r	Opens the file for reading. If the file doesn't exist, fopen() returns NULL
w	Opens the file for writing. If a file of the specified name doesn't exist, it is created. If a file of the specified name does exist, it is deleted without warning, and a new, empty file is created.
a	Opens the file for appending. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is appended to the end of the file.
r+	Opens the file for reading and writing. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is added to the beginning of the file, overwriting existing data.
w+	Opens the file for reading and writing. If a file of the specified name doesn't exist, it is created. If the file does exist, it is overwritten.
a+	Opens a file for reading and appending. If a file of the specified name doesn't exist, it is created. If the file does exist, new data is appended to the end of the file.

Different Steps to Open File

- Step 1 : Declaring FILE pointer.
- Firstly one needs pointer variable which can point to file.
- The syntax for declaring the file pointer is.
  - FILE *fp;
- Step 2 : Opening file hello.txt
  - fp = fopen ("filename","mode");

Closing a File

- fclose() function in File Handling.
- When a programmer opens a file in read or write mode then he performs appropriate operations on file and when file is no longer needed then he closes the file.
- FILE close will flush out all the entries from buffer.

Important

- After performing all Operations , FILE is no longer needed.
- File is closed using fclose() function.
- All information associated with file is flushed out of buffered.
- Closing file will prevent misuse of FILE.

Syntax : fclose()

- intfclose(FILE *fp);

Simple Example

```
#include<stdio.h>

int main()
{
FILE *fp;

fp = fopen("INPUT.txt","r")

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

fclose(fp); //File is no longer needed
}
```

Writing and Reading File Data

- A program that uses a disk file can write data to a file, read data from a file, or a combination of the two.
- One can write data to a disk file in three ways:
  - First way:One can use formatted output to save formatted data to a file.
  - One should use formatted output only with text-mode files.
- The primary use of formatted output is to create files containing text and numeric data to be read by other programs such as spreadsheets or databases.
- Second way: One can use character output to save single characters or lines of characters to a file.
- Although technically it's possible to use character output with binary-mode files, it can be tricky.
- One can write data to a disk file in three ways:
  - One should restrict character-mode output to text files.
  - The main use of character output is to save text (but not numeric) data in a form that can be read by C, as well as other programs such as word processors.
  - Third way: One can use direct output to save the contents of a section of memory directly to a disk file.
  - This method is for binary files only.
- Direct output is the best way to save data for later use by a C program.
- When one wants to read data from a file, he has the same three options: formatted input, character input, or direct input.
- The type of input one would use in a particular case depends almost entirely on the nature of the file being read.
- The data will be read in the same mode that it was saved in, but this is not a requirement.

Formatted File Input and Output

- Formatted file input/output deals with text and numeric data that is formatted in a specific way.
- It is directly analogous to formatted keyboard input and screen output done with the printf() and scanf() functions.

Formatted File Output

- Formatted file output is done with the library function fprintf().
- The prototype of fprintf() is in the header file STDIO.H, and it reads as follows:
  - int fprintf(FILE *fp, char *fmt, ...);
- The first argument is a pointer to type FILE.
- To write data to a particular disk file, pass the pointer that was returned when the file was opened with fopen().
- The second argument is the format string.
- The format string used by fprintf() follows exactly the same rules as printf().
- In other words, in addition to the file pointer and the format string arguments, fprintf() takes zero, one, or more additional arguments.
- This is just like printf(). These arguments are the names of the variables to be output to the specified stream.
- Remember, fprintf() works just like printf(), except that it sends its output to the stream specified in the argument list.

Formatted File Input

- For formatted file input, use the fscanf() library function, which is used like scanf() , except that input comes from a specified stream instead of from stdin.
- The prototype for fscanf() is:
  - Int fscanf(FILE *fp, const char *fmt, ...);
- The argument fp is the pointer to type FILE returned by fopen(), and fmt is a pointer to the format string that specifies how fscanf() is to read the input.
- The components of the format string are the same as for scanf().
- Finally, the ellipses (...) indicate one or more additional arguments, the addresses of the variables where fscanf() is to assign the input.

Character Input and Output

- When used with disk files, the term character I/O refers to single characters as well as lines of characters.
- The line is a sequence of zero or more characters terminated by the newline character.
- Use character I/O with text-mode files.

Character Input

- There are three character input functions: getc() and fgetc() for single characters, and fgets() for lines.
- The functions getc() and fgetc() are identical and can be used interchangeably.
- They input a single character from the specified stream.
- Here is the prototype of getc(), which is in STDIO.H:
  - int getc (FILE *fp);
- The argument fp is the pointer returned by fopen() when the file is opened.
- The function returns the character that was input or EOF on error.
- getc() was used in earlier programs to input a character from the keyboard.
- This is another example of the flexibility of C's streams--the same function can be used for keyboard or file input.

The getc() Function – Program

```
#include <stdio.h>
int main ()
{
  FILE *fp;
  int c;
  int n = 0;

  fp = fopen("file.txt","r");
  if(fp == NULL)
  {
    perror("Error in opening file");
    return(-1);
  }
  do
  {
    c = fgetc(fp);
    if( feof(fp) )
    {
       break ;
    }
    printf("%c", c);
  }while(1);

  fclose(fp);
  return(0);
}
```

The fgets() Function

- To read a line of characters from a file, use the fgets() library function.
- The prototype is: char *fgets(char *str, int n, FILE *fp);
- The argument str is a pointer to a buffer in which the input is to be stored, n is the maximum number of characters to be input, and fp is the pointer to type FILE that was returned by fopen() when the file was opened.
- When called, fgets() reads characters from fp into memory, starting at the location pointed to by str.
- Characters are read until a newline is encountered or until n-1 characters have been read, whichever occurs first.
- By setting n equal to the number of bytes allocated for the buffer str, one prevents input from overwriting memory beyond allocated space. (The n-1 is to allow space for the terminating \0 that fgets() adds to the end of the string).
- If successful, fgets() returns str.

```
#include <stdio.h>
int main()
{
  FILE *fp;
  char str[60];

  /* opening file for reading */
  fp = fopen("file.txt" , "r");
  if(fp == NULL) {
    perror("Error opening file");
    return(-1);
  }
  if( fgets (str, 60, fp)!=NULL ) {
    /* writing content to stdout */
    puts(str);
  }
  fclose(fp);

  return(0);
}
```

The putc() Function

- The library function putc() writes a single character to a specified stream.
- Its prototype in STDIO.H is as shown below: int putc(intch, FILE *fp);
- The argument ch is the character to output.
- As with other character functions, it is formally called a type int, but only the lower-order byte is used.
- The argument fp is the pointer associated with the file (the pointer returned by fopen() when the file was opened).
- The function putc() returns the character just written if successful or EOF if an error occurs.
- The symbolic constant EOF is defined in STDIO.H, and it has the value -1.
- Because no "real" character has that numeric value, EOF can be used as an error indicator (with text-mode files only).

```
#include <stdio.h>

int main ()
{
  FILE *fp;
  int ch;

  fp = fopen("file.txt", "w+");
  for( ch = 33 ; ch <= 100; ch++ )
  {
    fputc(ch, fp);
  }
  fclose(fp);

  return(0);
}
```

The fputs() Function

- To write a line of characters to a stream, use the library function fputs().
- This function works just like puts().
- The only difference is that with fputs() one can specify the output stream.
- Also, fputs() doesn't add a newline to the end of the string; one must explicitly include it.
- Its prototype in STDIO.H is:
  - char fputs(char *str, FILE *fp);
- The argument str is a pointer to the null-terminated string to be written, and fp is the pointer to type FILE returned by fopen() when the file was opened.
- The string pointed to by str is written to the file, minus its terminating \0.
- The function fputs() returns a non-negative value if successful or EOF on error.

```c
#include <stdio.h>

int main ()
{
  FILE *fp;

  fp = fopen("file.txt", "w+");

  fputs("This is c programming.", fp);
  fputs("This is a system programming language.", fp);

  fclose(fp);

  return(0);
}
```

Direct File Input and Output

- The direct file I/O most often when one has to save data to be read later by the same or a different C program.
- Direct I/O is used only with binary-mode files.
- With direct output, blocks of data are written from memory to disk.
- Direct input reverses the process: A block of data is read from a disk file into memory.
- For example, a single direct-output function call can write an entire array of type double to disk, and a single direct-input function call can read the entire array from disk back into memory.
- The direct I/O functions are fread() and fwrite().

The fwrite() Function

- The fwrite() library function writes a block of data from memory to a binary-mode file.
- Its prototype in STDIO.H is: int fwrite(void *buf, int size, int count, FILE *fp);
- The argument buf is a pointer to the region of memory holding the data to be written to the file.
- The pointer type is void; it can be a pointer to anything.
- The argument size specifies the size, in bytes, of the individual data items, and count specifies the number of items to be written.
- The argument fp is, of course, the pointer to type FILE, returned by fopen() when the file was opened.
- The fwrite() function returns the number of items written on success; if the value returned is less than count, it means that an error has occurred.

The fwrite() Function - Program

```
#include<stdio.h>
 int main ()
{
  FILE *fp;
  char str[] = "hello turbo c ";

  fp = fopen( "file.txt" , "w" );
  fwrite(str , 1 , sizeof(str) , fp );

  fclose(fp);

  return(0);
}
```

The fread() Function

- The fread() library function reads a block of data from a binary-mode file into memory.
- Its prototype in STDIO.H is: int fread(void *buf, int size, int count, FILE *fp);
- The argument buf is a pointer to the region of memory that receives the data read from the file.
- As with fwrite(), the pointer type is void.
- The argument size specifies the size, in bytes, of the individual data items being read, and count specifies the number of items to read.
- Note how these arguments parallel the arguments used by fwrite().
- Again, the sizeof() operator is typically used to provide the size argument.
- The argument fp is the pointer to type FILE that was returned by fopen() when the file was opened.
- The fread() function returns the number of items read; this can be less than count if end-of-file was reached or an error occurred.

The fread() Function – Program

```
#include <stdio.h>
#include <string.h>
int main()
{
  FILE *fp;
  char c[] = "this is a program";
  char buffer[20];

 /* Open file for both reading and writing */
 fp = fopen("file.txt", "w+");
  /* Write data to the file */
  fwrite(c, strlen(c) + 1, 1, fp);

  /* Seek to the beginning of the file */
 fseek(fp, 0, SEEK_SET);

   /* Read and display data */
  fread(buffer, strlen(c)+1, 1, fp);
   printf("%s\n", buffer);
  fclose(fp);

  return(0);
}
```

File Buffering: Closing and Flushing Files

- When one has done with using a file, one should close it using the fclose() function.
- The fclose() prototype is as shown below
  - int fclose(FILE *fp);
- The argument fp is the FILE pointer associated with the stream; fclose() returns 0 on success or -1 on error.
- When one closes a file, the file's buffer is flushed (i.e. written to the file).
- One can also close all open streams except the standard ones (stdin , stdout , stdprn, stderr , and stdaux) by using the fcloseall() function.
- Its prototype is as shown below:
  - int fcloseall(void);
- This function also flushes any stream buffers and returns the number of streams closed.
- When a program terminates (either by reaching the end of main() or by executing the exit() function), all streams are automatically flushed and closed.
- However, it's a good idea to close streams explicitly--particularly those linked to disk files.

The ftell() and rewind() Functions

- To set the position indicator to the beginning of the file, use the library function rewind().
- Its prototype, in STDIO.H, is:
  - void rewind(FILE *fp);
- The argument fp is the FILE pointer associated with the stream.
- After rewind() is called, the file's position indicator is set to the beginning of the file (byte 0).
- The rewind() is used to read some data from the beginning of the file again without closing and reopening the file.

The ftell() and rewind() Functions

- To determine the value of a file's position indicator, use ftell().
- This function's prototype, is located in STDIO.H and reads as follows:
  - long ftell(FILE *fp);
- The argument fp is the FILE pointer returned by fopen() when the file was opened.
- The function ftell() returns a type long that gives the current file position in bytes from the start of

the file (the first byte is at position 0).

- If an error occurs, ftell() returns -1 (of type long).

fseek() Function

- More precise control over a stream′s position indicator is possible with the fseek() library function.
- By using fseek(), one can set the position indicator anywhere in the file.
- The function prototype , in STDIO.H, is as follows:
  - int fseek(FILE *fp, long offset, int origin);
- The argument fp is the FILE pointer associated with the file.
- The distance that the position indicator is to be moved is given by its offset in bytes.
- The argument origin specifies the move•s relative starting point.
- There can be three values for origin, with symbolic constants defined in IO.H

feof() Function

- With a binary-mode stream, one can′t detect the end-of-file by looking for -1, because a byte of data from a binary stream could have that value, which would result in a premature end of input.
- Instead the library can be used.- function feof(), - which can be used for both binary- and text-mode files as shown below:
  - int feof(FILE *fp);
- The argument fp is the FILE pointer returned by fopen() when the file was opened.
- The function feof() returns 0 if the end of file fp hasn′t been reached, or a nonzero value if end-of-file has been reached.
- If a call to feof() detects end-of-file, no further read operations are permitted until a rewind() has been done, fseek() is called, or the file is closed and reopened.