SYLLABUS

Programming in C

Third Year (Fifth Semester)

Rationale : Computers play a vital role in present day life, more so, in the professional life of technician engineers. People working in the field of computer industry, use computers in solving problems more easily and effectively. In order to enable the students use the computers effectively in problem solving, this course offers the modern programming language C along with exposition to various applications of computers. The knowledge of C language will be reinforced by the practical exercises.

DETAILED CONTENTS

1. Algorithm and Programming Development

- 1.1 Steps in development of a program
- 1.2 Flow charts, Algorithm development
- 1.3 Programme Debugging
- 1.4 Basis of C programming

2. Program Structure

- 2.1 I/O statements, assign statements
- 2.2 Constants, variables and data types
- 2.3 Operators and Expressions
- 2.4 Standards and Formatted IOS
- 2.5 Data Type Casting

3. Control Structures

- 3.1 Introduction
- 3.2 Decision making with IF . statement
- 3.3 IF . Else and Nested IF
- 3.4 While and do-while, for loop
- 3.5 Break. Continue, goto and switch statements

4. Pointers

- 4.1 Introduction to Pointers
- 4.2. Address operator and pointers
- 4.3 Declaring and Initializing pointers,
- 4.4 Single pointer

5. Functions

- 5.1 Introduction to functions
- 5.2 Global and Local Variables
- 5.3 Function Declaration
- 5.4 Standard functions
- 5.5 Parameters and Parameter Passing
- 5.6 Call - by value/reference
- 5.7 Recursion

6. Arrays

- 6.1 Introduction to Arrays
- 6.2 Array Declaration, Length of array
- 6.3 Single and Multidimensional Array.
- 6.4 Arrays of characters
- 6.5 Passing an array to function
- 6.6 Pointers to an array

LIST OF PRACTICALS

1. Programming exercises on executing and editing a C program.
2. Programming exercises on defining variables and assigning values to variables.
3. Programming exercises on arithmetic and relational operators.
4. Programming exercises on arithmetic expressions and their evaluation.
5. Programming exercises on formatting input/output using printf and scanf and their return type values.
6. Programming exercises using if statement.
7. Programming exercises using if . Else.
8. Programming exercises on switch statement.
9. Programming exercises on do . while, statement.
10. Programming exercises on for . statement.
11. Programs on one-dimensional array.
12. Programs on two-dimensional array.
13. (i) Programs for putting two strings together.
    (ii) Programs for comparing two strings.
14. Simple programs using structures.
15. Simple programs using pointers.
16. Simple programs using union.

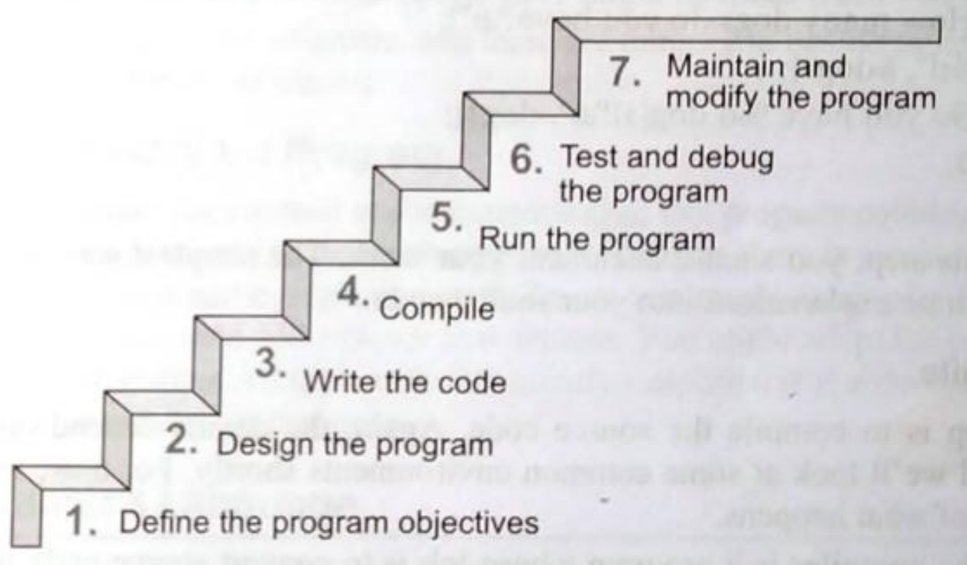
CONTENTS

1 Algorithm and Programming Development

Steps in Development of a Program

C, as you've seen, is a compiled language. If you are accustomed (havitual) to using a compiled language, such as Pascal or FORTRAN, you will be familiar with the basic steps in putting together a C program. However, if your background is in an interpreted language, such as BASIC, or in a graphical interface-oriented language, such as Visual Basic, or if you have no background at all, you need to learn how to compile. We'll look at that process soon, and you'll see that it is straightforward and sensible. First, to give you an overview of programming, let's break down the act of writing a C program into seven steps.

The seven Steps of Programming



Step 1 : Define the Program Objectives

Naturally enough, you should start with a clear idea of what you want the program to do. Think in terms of the information your program needs, the feats of calculation and manipulation the program needs to do, and the information the program should report back to you. At this level of planning, you should be thinking in general terms, not in terms of some specific computer language.

Step 2 : Design the Program

After you have a conceptual picture of what your program ought to do, you should decide how the program will go about it. What should the user interface be like? How should the program be organized? Who will the target user be? How much time do you have to complete the program?

You also need to decide how to represent the data in the program and, possibly, in auxiliary files,

as well as which methods to use to process the data. When you first learn programming in C, the choices will be simple, but as you deal with more complex situations, you'll find that these decisions require more thought. Choosing a good way to represent the information can often make designing the program and processing the data much easier.

Again, you should be thinking in general terms, not about specific code, but some of your decisions may be based on general characteristics of the language. For example, a C programmer has more options in data representation than, say, a Pascal programmer.

Step 3 : Write the Code

Now that you have a clear design for your program, you can begin to implement it by writing the code. That is, you translate your program design into the C language. Here is where you really have to put your knowledge of C to work. You can sketch your ideas on paper, but eventually you have to get your code into the computer. The mechanics of this process depend on your programming environment. We'll present the details for some common environments soon. In general, you use a text editor to create what is called a source code file. This file contains the C rendition of your program design.

Example of C Source Code :

```
#include <stdio.h>
int main (void)
{
int dogs;
printf("How many dogs do you have?\n");
scanf("%d", &dogs);
printf("So you have %d dog(s)!\n", dogs);
return 0;
}
```

As part of this step, you should document your work. The simplest way is to use C's comment facility to incorporate explanations into your source code.

Step 4 : Compile

The next step is to compile the source code. Again, the details depend on your programming environment, and we'll look at some common environments shortly. For now, let's start with a more conceptual view of what happens.

Recall that the compiler is a program whose job is to convert source code into executable code. *Executable code* is code in the native language, or *machine language*, of your computer. This language consists of detailed instructions expressed in a numeric code. As you read earlier, different computers have different machine languages, and a C compiler translates C into a particular machine language. C compilers also incorporate code from C libraries into the final program; the libraries contain a fund of standard routines, such as printf() and scanf(), for your use. (More accurately, a program called a linker brings in the library routines, but the compiler runs the linker for you on most systems.) The end result is an executable file containing code that the computer understands and that you can run. The compiler also checks that your program is valid C. If the compiler finds errors, it reports them to you and doesn't produce an executable file.

Step 5 : Run the Program

Traditionally, the executable file is a program you can run. To run the program in many common environments, including MS-DOS, Unix. Linux consoles, just type the name of the executable file. Other environments, such as VMS on a VAX, might require a run command or some other mechanism. Integrated development environments (IDEs), such as those provided for Windows and Macintosh environments, allow you to edit and execute your C program from within the IDE by selecting choices from a menu or by pressing special keys. The resulting program also can be run directly from the operating system by clicking or double-clicking the filename or icon.

Step 6 : Test and Debug the Program

The fact that your program runs is a good sign, but it's possible that it could run incorrectly. Consequently, you should check to see that our program does what it is supposed to do. You'll find that some of your programs have mistakes—*bugs*, in computer jargon. *Debugging* is the process of finding and fixing program errors. Making mistakes is a natural part of learning. It seems inherent to programming, so when you combine learning and programming, you had best prepare yourself to be reminded often of your fallibility. As you become a more powerful and subtle programmer, your errors, too, will become more powerful and subtle.

You have many opportunities to error. You can make a basic design error. You can implement good ideas incorrectly. You can overlook unexpected input that messes up your program. You can use C incorrectly. You can make typing errors. You can put parentheses in the wrong place, and so on. You'll find your own items to add to this list.

Fortunately, the situation isn't hopeless, although there might be times when you think it is.

The compiler catches many kinds of errors, and there are things you can do to help yourself track down the ones that the compiler doesn't catch.

Step 7: Maintain and Modify the Program

When you create a program for yourself or for someone else, that program could see extensive use. If it does, you'll probably find reasons to make changes in it. Perhaps there is a minor bug that shows up only when someone enters a name beginning with Zz, or you might think of a better way to do something in the program. You could add a clever new feature. You might adapt the program so that it runs on a different computer system. All these tasks are greatly simplified if you document the program clearly and if you follow sound design practices.

C is a Programmer's Language

In contrast, C was created, influenced, and field-tested by working programmers. The end result is that C gives the programmer what the programmer wants: few restrictions, few complaints, block structure, stand-alone functions, and a compact set of keywords.

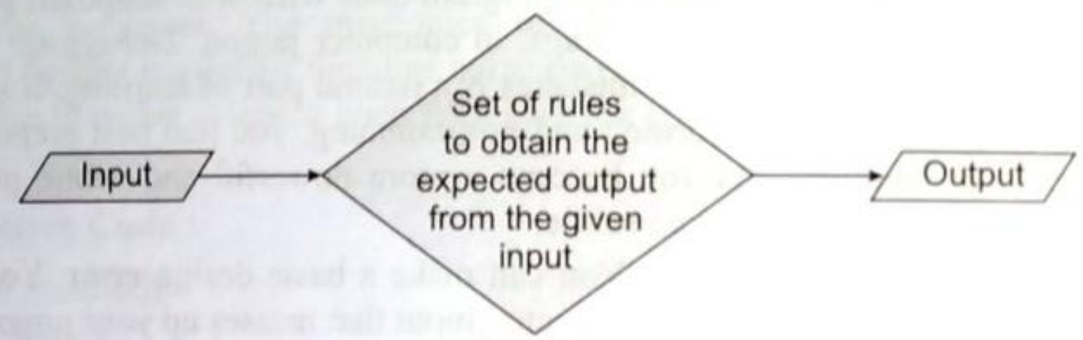
Initially, C was used for systems programming. A *systems program* forms a portion of the operating system of the computer or its support utilities, such as editors, compilers, linkers, and the like. As C grew in popularity, many programmers began to use it to program all the tasks because of its portability and efficiency. At the time of creation, C was a much longed-for, dramatic improvement in programming languages.

FEATURES OF C

C being popular language, all the instructions is given using simple English like statements. It uses a compiler which translates these instructions to binary, called object code. This object code is taken by linker who generates another binary coded program, called executable code. This is code which can be executable on the system.

Introduction to Algorithms

The word **Algorithm** means "a process or set of rules to be followed in calculations or other problem-solving operations". Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.



Algorithm

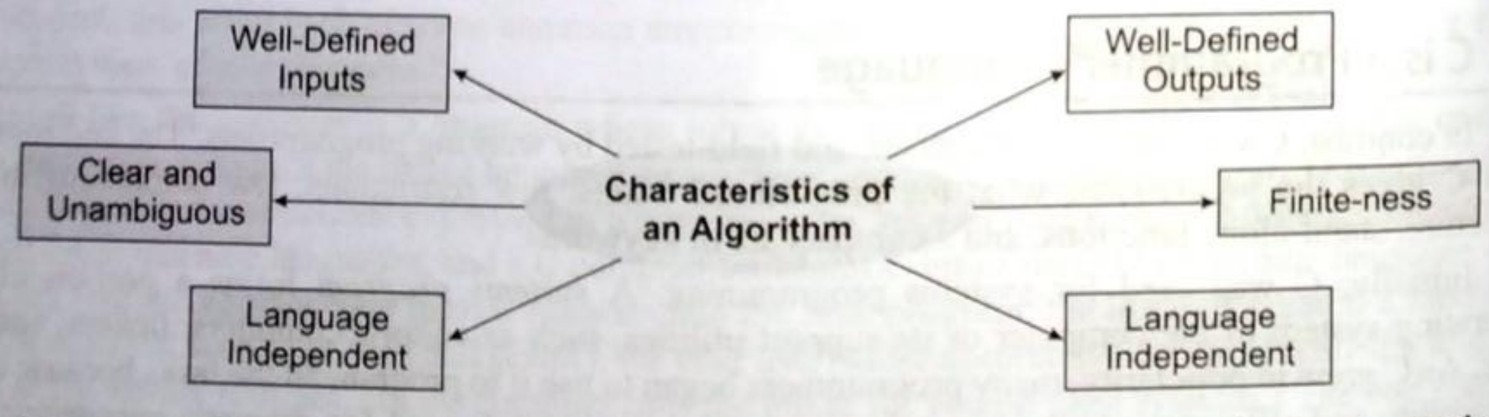
What is Algorithm?

It can be understood by taking an example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and execute them one by one, in the given sequence. The result thus obtained is the new dish cooked perfectly. Similarly, algorithms help to do a task in programming to get the expected output.

The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Characteristics of an Algorithm

Characteristics on an Algorithm



As one would not follow any written instructions to cook the recipe, but only the standard one. Similarly, not all written instructions for programming is an algorithm. In order for some instruction to be an algorithm, it must have the following characteristics :

- **Clear and Unambiguous :** Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

- **Well-Defined Inputs :** If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs :** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness :** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible :** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent :** The Algorithm designed must be language-independent, *i.e.* it must be just plain instructions
- That can be implemented in any language, and yet the output will be same, as expected.

How to Design an Algorithm

In order to write an algorithm, following things are needed as a pre-requisite :

1. The **problem** that is to be solved by this algorithm.
2. The **constraints** of the problem that must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem is solved.
5. The **solution** to this problem, in the given constraints.

Then the algorithm is written with the help of above parameters such that it solves the problem.

- **Example :** Consider the example to add three numbers and print the sum.

Step 1 : Fulfilling the pre-requisites

As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.

1. **The problem that is to be solved by this algorithm :** Add 3 numbers and print their sum.
2. **The constraints of the problem that must be considered while solving the problem :** The numbers must contain only digits and no other characters.
3. **The input to be taken to solve the problem :** The three numbers to be added.
4. **The output to be expected when the problem the is solved :** The sum of the three numbers taken as the input.
5. **The solution to this problem, in the given constraints :** The solution consists of adding the 3 numbers. It can be done with the help of'+' operator, or bit-wise, or any other method.

Step 2 : Designing the algorithm

Now let's design the algorithm with the help of above pre-requisites :

Algorithm to add 3 numbers and print their sum :

1. START
2. Declare 3 integer variables num 1, num 2 and num 3.
3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
5. Add the 3 numbers and store the result in the variable sum.
6. Print the value of variable sum
7. END

Step 3 : Testing the algorithm by implementing it

In order to test the algorithm; let's implement it in C language.

Program :

```
// C program to add three numbers
// with the help of above designed algorithm
  # include <stdio.h>
  int main()
  {
// Variables to take the input of the 3 numbers
  int num1, num2, num3;
// Variable to store the resultant sum
  int sum;
// Take the 3 numbers as input
  printf("Enter the 1st number:");
  scanf("%d", &num1);
  printf("%d\n", num1);
  printf("Enter the 2nd number: ");
  scanf("%d", &num2);
  printf("%d\n", num2);
  printf("Enter the 3rd number: ");
  scanf("%d", &num3);
  printf("%d\n", num3);
// Calculate the sum using + operator and store it in variable sum
  sum = num1 + num2 + num3;
// Print the sum
  printf("\nSum of the 3 numbers is: %d", sum);
  return 0;
  }
```

Output :

Enter the 1st number : 2
Enter the 2nd number : 3
Enter the 3rd number : 5
Sum of the 3 numbers is : 10

Examples of Algorithms :

Write an algorithm to add two numbers entered by the user.

Step 1 : Start
Step 2 : Declare variables num1, num2 and sum.
Step 3 : Read values num1 and num2.
Step 4 : Add num1 and num2 and assign the result to sum.
    sum ← num1 + num2
Step 5 : Display sum

Step 6 : Stop

Write an algorithm to find the largest among three different numbers entered by the user.

Step 1 : Start
Step 2 : Declare variables a,b and c.
Step 3 : Read variables a,b and c.
Step 4 : If a > b
    If a > c
        Display a is the largest number.
    Else
        Display c is the largest number.
    Else
    If b > c
        Display b is the largest number.
    Else
        Display c is the greatest number.
Step 5 : Stop

Write an algorithm to find all roots of a quadratic equation $ax^2 + bx + c = 0$.

Step 1 : Start
Step 2 : Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3 : Calculate discriminant
    D ← $b^2 - 4ac$
Step 4 : If D > 0
    r1 ← $(-b + \sqrt{D})/2a$
    r2 ← $(-b - \sqrt{D})/2a$
    Display r1 and r2 as roots.
Else
    Calculate real part and imaginary part
    rp ← b/2a
    ip ← $\sqrt{(-D)} / 2a$
    Display rp+j(ip) and rp-j(ip) as roots
Step 5 : Stop

Write an algorithm to find the factorial of a number entered by the user.

Step 1 : Start
Step 2 : Declare variables n, factorial and i.
Step 3 : Initialize variables
    factorial ← 1
    i ← 1
Step 4 : Read value of n
Step 5 : Repeat the steps until i = n
  5.1 : factorial ← factorial*!

5.2 : i ← i + 1

Step 6 : Display factorial

Step 7 : Stop

Write an algorithm to check whether a number entered by the user is prime or not.

Step 1 : Start

Step 2 : Declare variables n, i, flag.

Step 3 : Initialize variables

flag ← 1 i ← 2

Step 4 : Read n from user.

Step 5 : Repeat the steps until i<(n/2)

5.1 If remainder of n÷i equals 0

flag ← 0

Go to step 6

5.2 i ← i + 1

Step 6 : If flag = 0

Display n is not prime

else

Display n is prime

Step 7 : Stop

Write an algorithm to find the Fibonacci series till term ≤ 1000.

Step 1 : Start

Step 2 : Declare variables first_term,second_term and temp.

Step 3 : Initialize variables first_term ← 0 second_term ← 1

Step 4 : Display first_term and second_term

Step 5 : Repeat the steps until second_term ≤ 1000

5.1 : temp ← second_term

5.2 : second_term ← second_term + first_term

5.3 : first_term ← temp

5.4 : Display second_term

Step 6 : Stop

Implementation Phase

1. **Write code** : translate the algorithm into a programming language.
2. **Test** : have the computer follow the steps (execute the program).
3. **Debug** : check the results and make corrections until the answers are correct.
4. **Use the program.**

Flow Chart

A flow chart is a graphical or symbolic representation of a process. Each step in the process is represented by a different symbol and contains a short description of the process step. The flow chart symbols are linked together with arrows showing the process flow direction.

Nowadays, flowcharts play an extremely important role in displaying information and assisting reasoning. They help us visualize complex processes, or make explicit the structure of problems and tasks. A flowchart can also be used to define a process or project to be implemented.

Flowchart Symbols

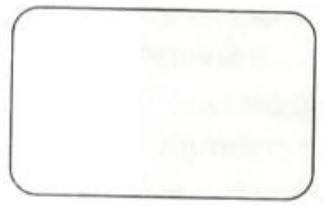
Different flowchart shapes have different conventional meanings. The meanings of some of the more common shapes are as follows:

Terminator

The terminator symbol represents the starting or ending point of the system.

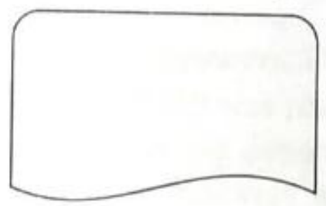
Process

A box indicates some particular operation.



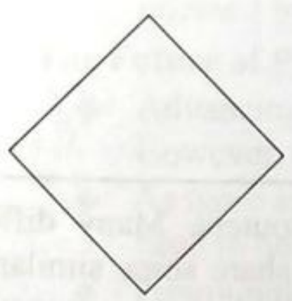
Document

This represents a printout, such as a document or a report.

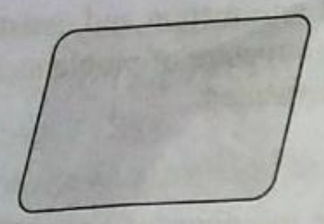


Decision

A diamond represents a decision or branching point. Lines coming out from the diamond indicates different possible situations, leading to different sub-processes.

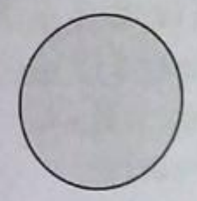


It represents information entering or leaving the system. An input might be an order from a customer. Output can be a product to be delivered.



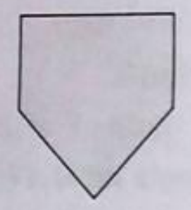
On-Page Reference

This symbol would contain a letter inside. It indicates that the flow continues on a matching symbol containing the same letter somewhere else on the same page.



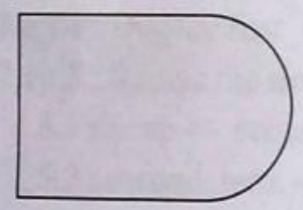
Off-Page Reference

This symbol would contain a letter inside. It indicates that the flow continues on a matching symbol containing the same letter somewhere else on a different page.



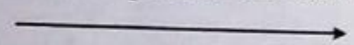
Delay or Bottleneck

Identifies a delay or a bottleneck.



Flow

Lines represent the flow of the sequence and direction of a process.



When to Draw

Programming Languages and Their Uses

Programmers use programming languages to communicate with computers. Many different languages exist, and each one has its own unique features, though they all share some similarities. Because each language is different, each may be best suited for a certain purpose or purposes within certain industries. Some programming languages are used to create programs to solve problems or interpret data. Other programming languages are more suitable for making create software or apps that entertain. With a strong need for unique and diverse programming languages, it is virtually impossible to create a single universal programming language that meets all needs. Programming languages are

often revised and even combined with other languages over time, evolving to meet our changing technological needs. "Computers understand instructions that are written in a specific syntactical form called a programming language. A programming language provides a way for a programmer to express a task so that it could be understood and executed by a computer."

Programming Languages

- Python is an open-source programming language used by software engineers and back-end Web developers. Python is well-suited for scientific computing, and it is relatively simple to learn.
- Java is prevalent in Web-based development, and it was created in 1995. Many companies in the health sciences, education, and finance industries use Java. Java enables the downloading of applets from websites, which enable browsers to perform additional functions.
- Ruby is an open-source scripting language that coders can use independently or in conjunction with Ruby on Rails. NASA uses Ruby in its work with simulations.
- HTML is used extensively in Web development. HTML is the code that serves as the foundation of Web pages, allowing people to create and structure electronic documents for viewing online.
- JavaScript is used by Web developers and software engineers to manipulate page elements to make them more engaging. JavaScript enhances HTML, and it is embedded in most Internet browsers.
- C is a middle-level programming language used by software developers and systems analysts. Programmers use C to create applications that integrate with operating systems.
- C++, developed in 1983, is another middle-level programming language and works as an extension of C. Programmers use C++ to create games, graphics, and office applications.
- C# is a programming language used by software engineers who create applications designed to work with Windows operating systems. C# shares similarities with Java.
- Objective-C is an object-oriented language used by mobile developers and software engineers. Developers creating iOS and OS X utilities often use Objective-C.
- PHP was released to the public in 1995. Developers use PHP as an open-source language to create dynamic Web pages. Widely used platforms such as Word Press and Drupal work cohesively with PHP.
- SQL enables programmers to create, read, update, and delete information in a database. Companies use SQL to gather data.
- Apple uses the Swift programming language to create and maintain iOS and OS X applications. Swift 2 is a secondary open-source programming language more recently released by Apple.

The Future of Programming Languages

- Advancing technology promises that programming languages will continue to evolve. However, predicting the future of programming can be challenging.
- As more appliances and devices are designed to operate with a computer chip, software will need to be maintained regularly to keep it up to date and functioning correct!).
- Programmers continually face the challenges of protecting devices from viruses and developing applications that allow users to use their devices cohesively.
- The newest programming languages will be faster and more intuitive with fewer errors and issues. For example, R is one of the most recent programming languages and was designed by statisticians for data analysis.

Uses of Programming Languages

Web Development

If you're interested in building websites there are two intertwining parts to look into. First, there's front-end development, which is the part of web development that creates the application that runs on your browser and adjusts the styling, the colors, the interactions. It's basically concerned with what the user of a website sees. You are reading this blog on some screen which is shown to you by front-end code. Front-end basics start with HTML and CSS with use of JavaScript. Javascript has become one of the most dominant languages in the last few years for front-end work.

The other part for creating websites is **back-end development**, which is related to the server, the computer that runs the website software and serves it to the world. It's mostly concerned with routing, which pages to deliver to the user when they visit a certain URL, it also communicates with the database that stores the website's information and sends this data over to the user. Back-end development is where the magic happens and there are many options to choose from when it comes to a programming language, you can stick to Javascript just like in front-end development, or go with PHP, Ruby, C#, Elixir, Python, Erlang.

Game Development

Game development is one of the most interesting tracks there is, many developers enjoy it and there are developers who develop games just for having fun. Creating games requires what's called a game engine, which is a software that is used as the infrastructure for building the game and defines what the game has and what it can do. If you're familiar with Epic Games and Fortnight, Epic Games is, in fact, a game engine and Fortnight is built upon it. The languages used in game development are mostly C++, C# since it requires a lot of memory optimization and fast performance to create rich graphics. It's not limited to C++ and C#, however, and it kind of is about which engine you're using and which platform you're targeting, Lua and Java are also very famous candidates in this industry.

Mobile Development

Creating mobile applications is a little tricky, as there is more than one operating system for mobiles and the different operating system would require different languages for these applications. An operating system is the piece of software on your device that is responsible for dealing with the hardware of this device, it's the layer that sits between the application you create and the hardware, whether it's a mic or a touchscreen or GPS. The most two common operating systems are Android and IOS. Android is most commonly used in Samsung while IOS is used in Apple. To create Android apps, you'd need either Java or Kotlin, and for creating IOS applications you'd need Objective-C or Swift. Recenth. it became possible to create mobile applications for both Android and IOS using Javascript or Dart.

Programming Techniques

Software designing is very anesthetic phase of *software development cycle*. The beauty of heart, skill of mind and practical thinking is mixed with system objective to implement design.

The *designing process* is not simple, but complex, cumbersome and frustrating with many curves in the way of successful design.

Here are some approaches :

- Structural Programming
- Top Down Designing
- Object Oriented Programming
- Modular Designing
- Bottom Up Designing

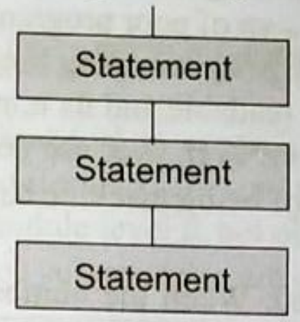
1. Structured Programming

This is the first programming approach used widely in beginning. Professor Edsger Wybe Dijkstra (1960) coins the term Structural Programming. Italian computer scientist C. Bohm and G. Jacopini (1966) give the basic principal that supports this approach. The structured programming movement started in 1970, and much has been written about it. It is often regarded as "goto-less" programming, because it is avoided by programmers.

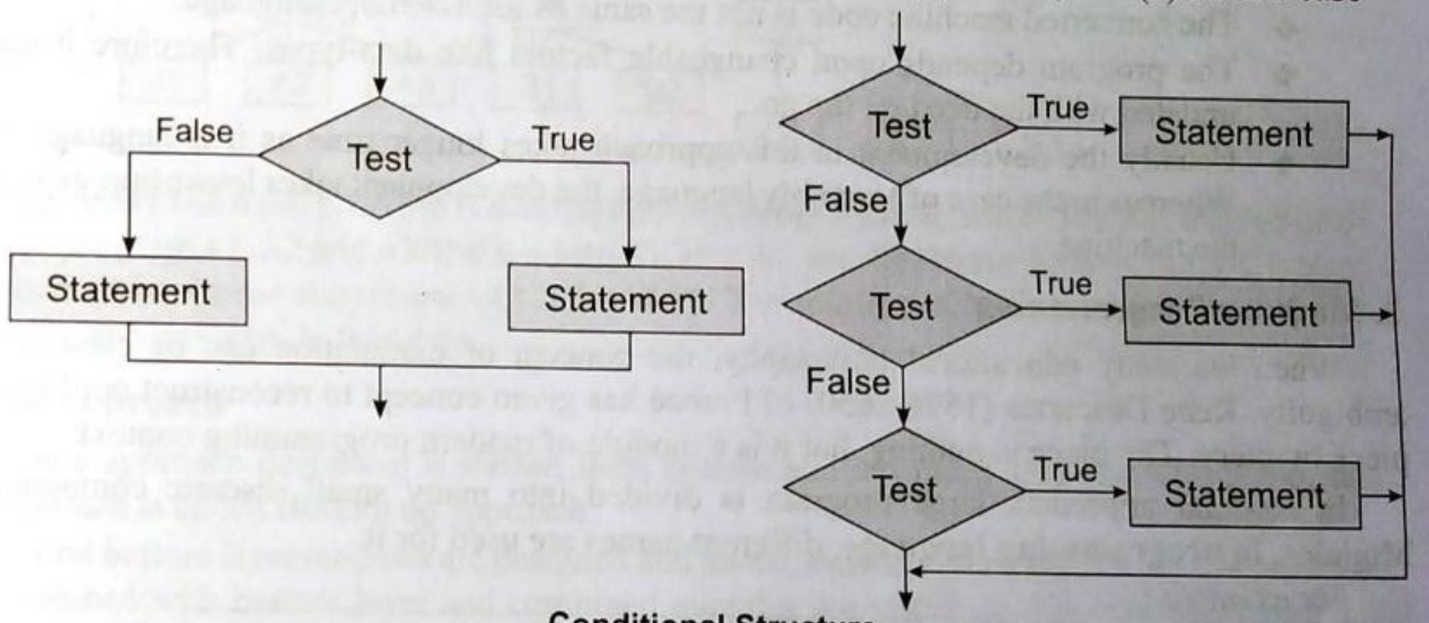
The program is divided into several basic structures. These structures are called building blocks.

These are following :

(a) **Sequence Structure :** This module contains program statements one after another. This is a very simple module of Structured Programming.

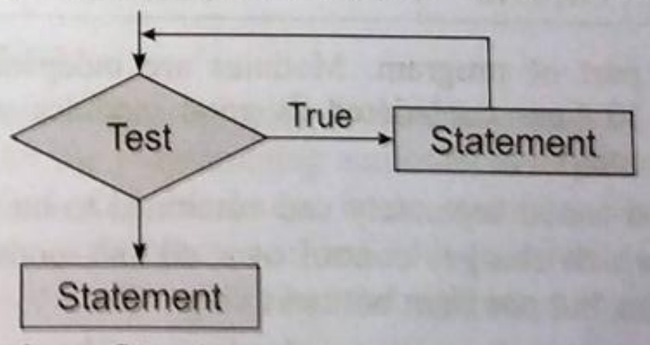


(b) **Selection or Conditional Structure :** The Program has many conditions from which correct condition is selected to solve problems. These are (a) if-else (b) else-if, and (c) switch-case



Conditional Structure

(c) **Repetition or loop Structure :** The process of repetition or iteration repeats statements blocks several times when condition is matched, if condition is not matched, looping process is terminated. In C, (a) goto, (b) for (), (c) do, (d) do - while are used for this purpose.



Loop Structure

Advantage :

- Problem can be easily described using Flowchart and flowchart can be easily coded into program because the nature of this technique is like as flowchart.
- The program is easily coded using modules.
- The testing and debugging is easy because testing and debugging can be performed module-wise.
- Program development cost low.
- Higher productivity, high quality program production.
- Easy to modify and maintain
- It is called **"gototess"** *programming* because use of *goto* for unconditional branching is strongly avoided. The goto is a sign of poor program design, so many designing concepts are not favoring it but it is used in all **programming language** widely. When more goto is used in program, program become less readable and its impact is negative in program functionality. So it is saying about goto:"Never, ever, ever use goto! It is evil".
- Mainly problem based instead of being machine based.

Disadvantage :

- More memory space is required. When the numbers of modules are out of certain range, performance of program is not satisfactory.
- Since it is machine -independent, so it takes time to convert into machine code.
- The converted machine code is not the same as for assembly language.
- The program depends upon changeable factors like data-types. Therefore it needs to be updated with the need on the go.
- Usually the development in this approach takes longer time as it is language-dependent. Whereas in the case of assembly language .the development takes lesser time as it is fixed for the machine.

2. Modular Programming

When we study educational philosophy, the concept of modulation can be clear without any ambiguity. Rene Descartes (1596-1650) of France has given concept to reconstruct our knowledge by piece by piece. The piece is nothing, but it is a module of modern programming context.

In modular approach, large program is divided into many small discrete components called Modules. In **programming language**, different names are used for it.

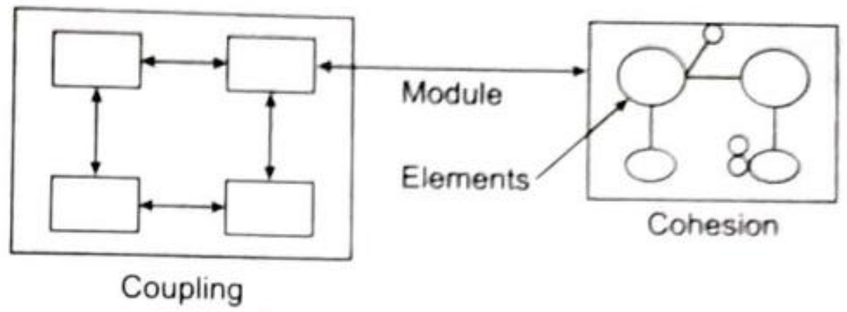
For example :

Q-basic, Fortran	Subroutine
Pascal	Procedure or Function
C, C+, C#, Java	Function

It is logically separable part of program. Modules are independent and easily manageable. Generally modules of 20 to 50 lines considered as good modules when lines are increased, the controlling of module become complex.

Modules are debugged and tested separately and combined to build system. The top module is called root or boss modules which charges control over all sub-modules from top to bottom. The control flows from top to bottom, but not from bottom to top.

The evaluation of modeling is called coupling and cohesion. The module coupling denotes number of interconnections between modules and module cohesion shows relationship among data or elements within a module.



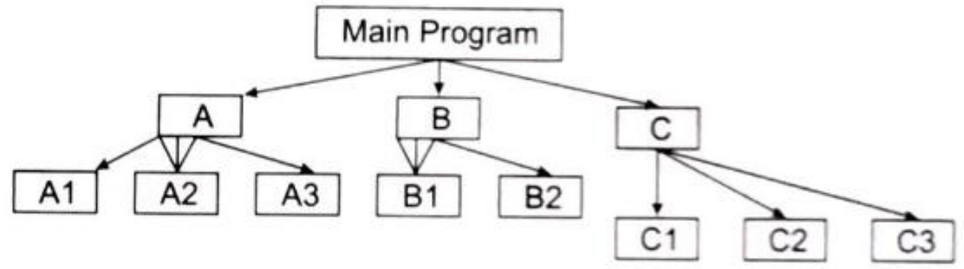
Modular Programming

3. Top down Approach

(a) The large program is divided into many small **module** or subprogram or function or procedure from top to bottom.

(b) At first supervisor program is identified to control other sub modules. Main modules are divided into sub modules, sub-modules into sub- sub- modules. The decomposition of modules is continuing whenever desired module level is not obtained.

(c) Top module is tested first, and then sub-modules are combined one by one and tested.



- **Example :** The main program is divided into sub-program A, B, and C. The A is divided into subprogram A1, A2 and A3.The B is into B1. and B2. Just like these subprograms, C is also divided into three subprogram C1. C2 and C3. The solution of Mam program is obtained from sub program A. B and C.

4. Bottom up Approach

- In this approach designing is started from bottom and advanced stepwise to top. So, this approach is called Bottom up approach.
- At first bottom layer modules are designed and tested, second layer modules are designed and combined with bottom layer and combined modules are tested. In this way, designing and testing progressed from bottom to top.
- In software designing, only pure top down or Bottom up approach is not used. The hybrid type of approach is recommended by many designers in which top down and bottom up, both approaches are utilized.

5. Object oriented programming

In the object-oriented programming, program is divided into a set of objects. The emphasis given on objects, not on procedures. All the programming activities revolve around objects. An object is a real world entity. It may be airplane, ship, car, house, horse, customer, bank Account, loan, petrol, fee, courses, and Registration number etc. Objects are tied with functions. Objects are not free for walk without leg of functions. One object talks with other through earphone of functions. Object is a boss but captive of functions.

Features of Object oriented Language

- The program is decomposed into several objects. In this language, emphasis is given to the objects and objects are central points of programming. All the activities are object centered.
- Objects occupy spaces in memory and have memory address like as records in PASCAL and structure in C language.
- Data and its functions are encapsulated into a single entity.
- Reusability: In C++, we create classes and these classes have power of reusability. Other programmers can use these classes.
- It supports bottom up approach of programming. In this approach designing is started from bottom and advanced stepwise to top.

Some technical terms supporting object-oriented languages are

(i) Abstraction : The abstraction is an important property of OOP. The use of essential features over less essential features is called abstraction. The following examples will help to understand abstraction.

- **Example :** The computer operators know only to operate computer, but they are unaware to internal organization of computer.

In OOP, there are many devices used for data abstraction such as class, encapsulation, data hiding etc.

(ii) Class : A class is a collection of similar objects. Objects are members of class. Once a class is declared, its many members can be easily created in programs. The class binds attributes (data and functions or methods) of member objects.

Examples :

```
Class employee
{
char name[30];
float basic;
void getdata();
void show();
};
```

(iii) Polymorphism : The ability to find in many forms is called **polymorphism** (Poly: many, **Morphe:** shape / form). For instance, + is mathematical operator, it concatenates two strings and give sum of two digits (numbers). Here, operator + has different behavior for numerical data and strings. Just like it, once declared function has different meaning that is called **function overloading**. If operator has different meaning, it is called operator overloading.

(iv) Encapsulation : The encapsulation is a very striking feature of OOP in which data and function is bound into single unit. Array, records, structure are also example of low level encapsulation but term encapsulation is mostly used in object oriented language. The data and function are encapsulated into class. External world or external function cannot access the data. It can be accessed by its own function or method encapsulated with it into class. It hides private elements of objects behind public interface.

(v) Inheritance : Inheritance is a hierarchy of class in which some properties of base class is transferred to derived class. The main types of inheritance are:

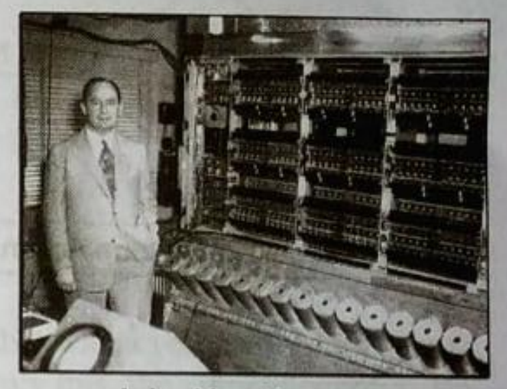
(a) **Single Inheritance :** A derived class (child class or sub class) of single base (super or parent) class is called single Inheritance.

(b) **Multiple Inheritances :** A derived class of multiple base classes is called Multiple Inheritance.

(c) **Multilevel Inheritance :** When derived class is derived from another derived class, such type of inheritance is called Multilevel Inheritance.

Introduction to Programming History

Without a program, a computer has no purpose. It becomes an inert mass, a way to keep a door open when the wind blows.



John Von Neumann

In 1952, a very clever man named John Von Neumann was working on a rather primitive computer that was programmed by moving wires about on a large plug-board.

It occurred to him that the program could be stored within the machine, along with the data, in what we now call "random-access memory."

This research removed the final obstacle to large, complex programs, the ability to switch tasks quickly, and the logical certainly that a program above a certain complexity level cannot be proven to be bug-free.

It can fairly be argued that a computer program is a product of pure intellect, has no physical existence, and therefore cannot meaningfully be secured against theft. To summarize, a computer is a machine whose sole purpose is to faithfully carry out a computer program's instructions. It is no more than an input/output device to support the intellectual goals of a computer program (and a programmer).

The computer's higher purpose resides in its program. This is why, as time passes, computers become less expensive and programs become more expensive.

It is why the richest man in the world is, not a builder of computers, but a builder of programs.

It is why, over time, computer scientists have come to focus more on programming issues and less on hardware issues.

It is why, when a government agency decides to upgrade its computers, it is almost always the case that, after billions have been spent, they discover the old software will not run on the new hardware, and they abandon the project.

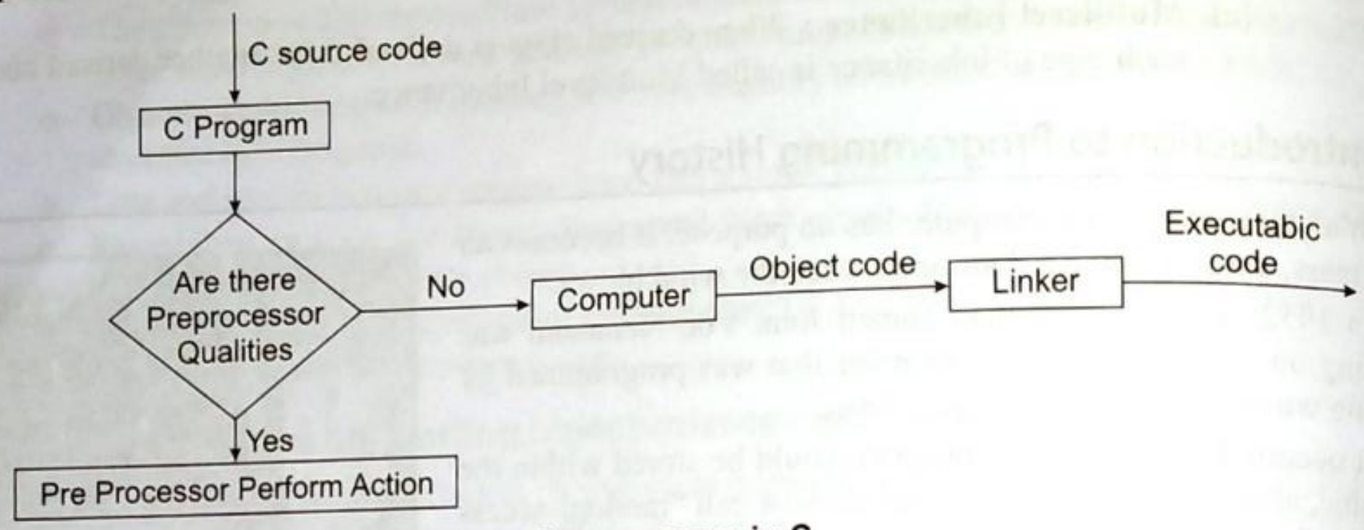
C Pre-processors

As the name suggests Pre-processors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C . Let us have a look at these steps before we actually start learning about Pre-processors.

You can see the intermediate steps in the above diagram. The source code written by programmers is stored in the file program.c. This file is then processed by pre-processors and an expanded source code file is generated named program. This expanded file is compiled by the compiler and an object code file is generated named program .obj. Finally, the linker links this object code file to the object code of the library functions to generate the executable file program.exe.

Preprocessor programs provide preprocessor directive which tell the compiler to pre-process the source code before compiling. All of these preprocessor directive begin with a '#'(hash) symbol. This ('#') symbol at the beginning of a statement in a c/c+ program indicates that it is a pre-processor directive.we can place these preprocessor directives anywhere in our program. examples of some

preproceeor directives are: #include, #define, # ifndef etc. there are 4 types of preprocessor directive :



Preprocessor in C

1. File inclusion
2. Conditional compilation
3. Other directive

Let us now learn about each of these directives in details.

Macros : macro are a piece of code in a program which is given some name, when ever this name is encountered by the compiler the compiler replaces the name with the actual piece of code. The '#define' directive is used to define a macro. Let us now understand the micro definition with the help of a program:

```
#include <stdio.h>
// macro definition
#define LIMIT 5
int main ()
{
   for (int i = 0; i < LIMIT; i++)
   {
      printf("%d \n",i);
   }
return 0;
```

Output :

```
0
1
2
3
4
```

In the above program , when the compiler executes the world LIMIT it replace it with 5.

The word 'LIMIT' in the macro definition is called a macro template and '5' is macro expansion.

- **Note :** there is no semi-colon(',') at the end of macro definition. Macro definition do not need a semi-colon to end.

Macros with arguments : we can also pass arguments to macros. Macros defined with arguments works similarly as functions. Let us understand this with a program,

```
#include <stdio.h>
// macro with parameter
#defme AREA(1, b) (1 * b)
int main()
{
   int 11 = 10,12 = 5, area;
   area = AREA(11,12);
   printf("Area of rectangle is: %d", area);
   return 0;
}
```

Output:

Area of rectangle is : 50 : We can see from the above program that whenever the compiler finds AREA(l,b) in the program it replaces it with the statement (l*b). not only this, the values passed to the macro template AREA(l,b) will also be replaced in the statement (l*b). therefore AREA(10,5) will be equal to 10*5.

File inclusion : This types of preprocessor directive tells the compiler to include a file in the source code program.there are two types of files which can be included by the user in the program.

1. **Header file or standard files :** these files contains definition of pre-defined functions like printf(), scanf() etc. these file must be included for working with these functions.different function are declared in different header files.
   For example statdard I/O functions are 'iostream' file whereas functions which perform string operations are in 'string' file.
   Syntax :
   #include<file name>
   Where file name is the name of file to be included.the '<' and '>' brackets tells the compiler to look for the file in standard directory.
2. **User defined files :** When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These file can be included as :
   #include "filename"

Conditional compilation : conditional compilation directive are types of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions. This can be done with the help of two preprocessing commands 'ifdef' and 'endif.

syntax :

```
#ifdef macro name
statement1;
statement2;
statement3;
.....;
.....;
statement N;
```

```
#endif
```

If the macro with name as 'macroname' is defined then the block of statement will execute normally but it it not defined, the complier will simply skip this block of statements.

Other directives: apart from the above directives there are two more directives which are not commonly used. These are:

1. **#undef Directive :** The #undef directive is used to undefine and existing macro
   This directive works as :
   #undef LIMIT
   Using this statemnet will undefine the existing macro LIMIT. After this statement every "#ifdef LIMIT" statement will evaluate to false.
2. **#pragma Directive :** This directive is a special purpose directive end is used to turn on or off some feathres. This type of directives are compiler-specific *i.e.*, they very. from complier to complier. Some of the #pragma directives are discussed below :

(i) #pragma startup and #pragma exit : These directives helps us to specify the functions that are needed to run before program startup (before the control passes to main()) and just before program exit (just before the fcontrol returns from main()).

- **Note :** Below program will not work with GCC compliers.

Look at the below program :

```
#include <stdio.h>
void func1();
void func2();
#pragma startup func1
//pragma exit func2
void func1 ()
{
   printf("Insidefunc1()\n");
}
void func2 ()
{
printf("Inside func2()");
}
int main()
{
   void func1();
   void func2();
   printf("Inside main()\n");
      return 0;
}
```

Output :

```
Inside func1()
Inside main()
```


The above code will produce the output as given below when run onGCC compliers :

```
Inside main()
```

This happens because GCC does nto supports #pragma startup or exit.

However you can use the below code for a similar output on GCC compliers.

```
#include <stdio.h>
void func1();
void func2();
void_attribute_((constructor)) func1();
void_attribute_((destructor)) func2();
void func1()
{
  printf("Insidefunc1()\n");
}
void func2()
{
  printf("Inside func2()\n");
}
int main()
{
  printf("Inside main()\n");
     return 0;
}
```

(ii) #pragma warn Directive : This directive is used to hide the warning message which are displayed during compilation.

We can hide the warnings as shown below :

1. **#pragma warn-rvl :** The directive hides those warning which are raised when a function which is supposed to return a value does not returns a value.
2. **#pragma warn-par :** This directive hide those warning which are raised when a function does not use the parameters passed to it.
3. **#pragma warn-rch :** The directive hides those warning which are raised when a code is unreachable. For example : any code written after the *return* statement in a function is ureachable.

Debugging

A Software Application needs to be error-free before going out in the market. Customer satisfaction is of utmost importance for any organization and only a bug-free product can keep your customer happy.

Software programs undergo heavy testing, updating, troubleshooting, and maintenance during the development process. Usually, the software contains **errors and bugs**, which are routinely removed. **Debugging** is the process of **fixing a bug** in the software.

It refers to identifying, analyzing and removing errors. This process begins after the software fa to execute properly and concludes by solving the problem and successfully **testing the software**. B it is considered to be an extremely **complex** and **tedious** task because errors need to be resolved at stages of debugging.

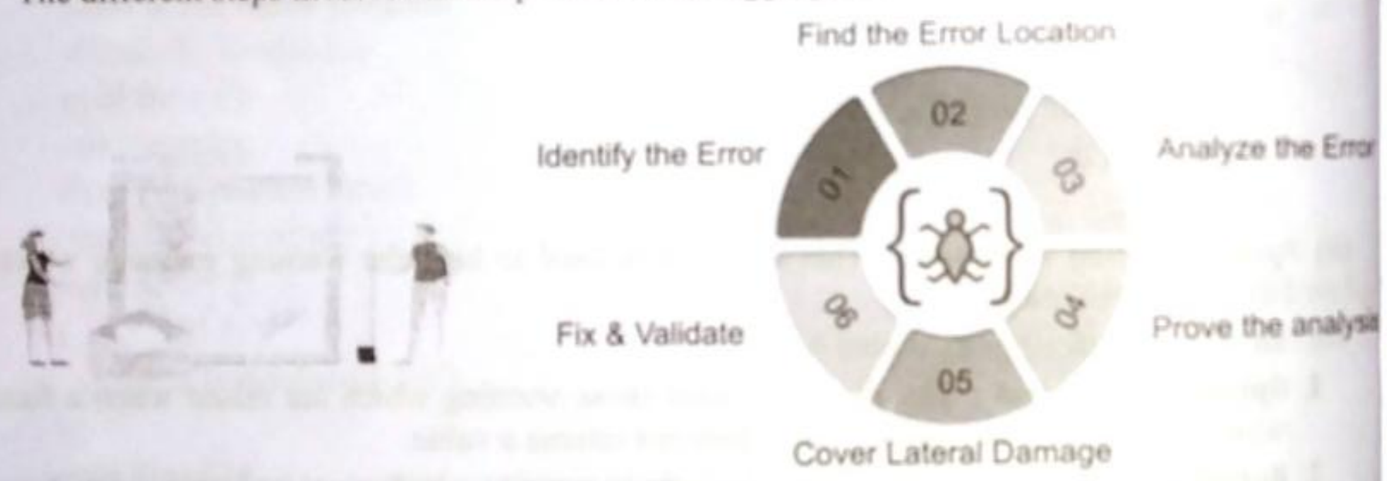
Why do we need Debugging?

The process of debugging begins as soon as the **code** of the **software** is written. Then, it contin in successive stages as code is combined with other units of programming to form a software produ Debugging has many benefits such as:

- It **reports an error condition immediately.** This allows earlier detection of an error a makes the process of software development stress-free and unproblematic.
- It also provides **maximum useful information** of data structures and allows **interpretation**.
- Debugging assists the developer in **reducing useless** and **distracting information.**
- Through debugging the developer can **avoid complex one-use testing code** to save time energy in software development.

Steps involved in Debugging

The different steps involved in the process of debugging are:



1. **Identify the Error :** A bad identification of an error can lead to wasted developing time. usual that production errors reported by users are hard to interpret and sometimes information we receive is misleading. It is import to identify the actual error.
2. **Analyze the Error :** In the third step, you need to use a boitoi > approach from the e location and analyze the code. This helps you in understanding the error. Analyzing a bug t\vo main goals, such as checking around the error for other errors to be found, and to make about the risks of entering any collateral damage in the fix.
3. **Prove the Analysis :** Once you are done analyzing the original bug. you need to find a more errors that may appear on the application. This step is about writing automated tests these areas with the help of a test framework.
4. **Cover Lateral Damage :** In this stage, you need to create or gather all the unit tests for code where you are going to make changes. Now, if you run these unit tests, they all sh pass.
5. **Fix & Validate :** The final stage is the fix all the errors and run all the test scripts to che they all pass.

Debugging Strategies

- It is important to study the system in depth in order to understand the system. It helps the debugger to construct different representations of systems that are to be debugged.
- **Backward analysis** of the problem traces the program backward from the location of failure message in order to identify the region of faulty code. You need to study the region of defect thoroughly to find the cause of defects.
- **Forward analysis** of the program involves tracking the program forward using breakpoints or print statements at different points in the program. It is important to focus on the region where the wrong outputs are obtained.
- You must use the **past experience** of the software to check for similar problems. The success of this approach depends on the expertise of the debugger.

Debugging Tools

Debugging tool is a computer program used to test and debug other programs. There are a lot of public domain software like **gdb** and **dbx** that you can use for debugging. Also, they offer console-based command-line interfaces. Some of the automated debugging tools include code-based tracers, profilers, interpreters, etc.

Here is a list of some of the widely used debuggers:

- **Radare2**
- **Valgrind**
- **WinDbg**

With this, we have come to the end of our article. I hope you understood what is debugging and the different stages involved in the process of debugging.

Advantages of Debugging

Below is the list of debugging advantages

- **Saves Time:** Performing debugging at the initial stage saves the time of software developers as they can avoid the use of complex codes in software development. It not only saves the time of software developers but also saves their energy.
- **Reports Errors:** It gives error report immediately as soon as they occur. This allows the detection of error at an early stage and makes the software development process a stress free.

Release bug-free software: By finding errors software, it allows developers to fix them before releasing them and provides bug-free software to the customers.

Compile

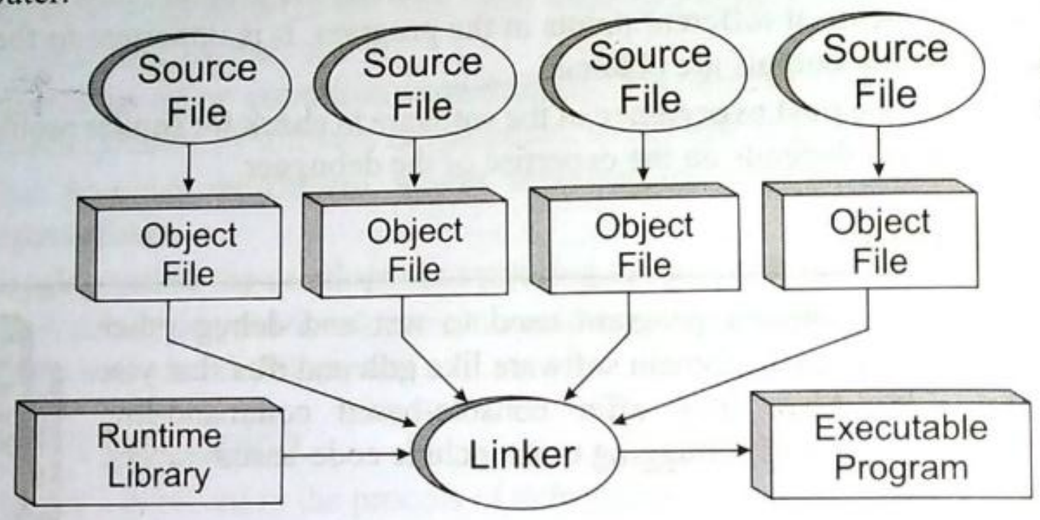
Compile is the process of creating an executable program from code written in a **compiled programming language.** Compiling allows the computer to run and understand the program without the need of the programming software used to create it. When a program is compiled it is often compiled for a specific platform (*e.g.*, IBM platform) that works with IBM compatible computers, but not other platforms (*e.g.*, Apple platform).

The first **compiler** was developed by Grace Hopper while working on the Harvard Mark I computer. Today, most high-level languages will include their own compiler or have toolkits available that can be used to

compile the program. Two popular compilers are Eclipse for Java and gcc command for C and C++. Depending on how big the program is, it should take a few seconds or minutes to compile. If no errors are encountered while being compiled, an executable file is created.

Compile Time

The **compile time** is the total time it takes a compiler to compile code into a program that can be run by the computer.



To transform a program written in a high-level programming language from source code into object code. Programmers write programs in a form called source code. Source code must go through several steps before it becomes an executable program. The first step is to pass the source code through a compiler, which translates the high-level language instructions into object code.

The final step in producing an executable program - after the compiler has produced object code - is to pass the object code through a linker. The linker combines modules and gives real values to all symbolic addresses, thereby producing machine code.

Basic Structure of C Program

A **C program** is divided into different sections. There are six main sections to a basic c program. The six sections are :

- ❖ Documentation
- ❖ Definition
- ❖ Main functions
- ❖ Link
- ❖ Global Declarations
- ❖ Sub programs

So now that the introduction is out of the way. let us jump to the main discussion. The whole code follows this outline. Each code has a similar outline. Now let us learn about each of this layer in detail.

Documentation
Link Section
Definition Section
Global Declaration Section
Main Function
Subprogram Section

Basic Structure of C Program

Basic Structure of C Program

Moving on to the next bit of this basic structure of a C program article,

Documentation Section

The documentation section is the part of the program where the programmer gives the details associated with the program. He usually gives the name of the program, the details of the author and other details like the time of coding and description. It gives anyone reading the code the overview of the code.

Comments are a way of explaining what makes a program. The compiler ignores comments and used by others to understand the code,

or

This is a comment block, which is ignored by the compiler. Comment can be used anywhere in the program to add info about the program or code block, which will be helpful for developers to understand the existing code in the future easily. Types of comments:

1. Single line comment
2. Multiline comment.

1. single line comment : Represented as // double forward slash, It is used to denote a single line comment. It applies comment to a single line only. It is referred to as C++-style comments as it is originally part of C++ programming.

For example :

```
//WAP to printf the name of your university.
```

Example :

```
// write a program to print the name of your university.
#include <stdio.h>
int main()
{
  // Single line Welcome user comment
  printf("Gopal Narayan Singh University");
  return 0;
}
```

2. Multi-line comment : Represented as /* **any_text** */,start with forward slash and asterisk (/*) and end with asterisk and forward slash (*/).

It is used to denote multi-line comment. It can apply comment to more than a single line. It is referred to as C-Style comment as it was introduced in C programming.

```
/* comment starts
.....
.....
* File Name: gnsu.c
* Author: Mr. xyz
* description: a program to display hello world no input needed
*/
```

Link Section

This part of the code is used to declare all the header files that will be used in the program.

This leads to the compiler being told to link the header files to the system libraries.

This is a preprocessor command. That notifies the compiler to include the header file stdio.h in the program before compiling the source-code.

A Header file is a collection of built-in (readymade) functions, which we can directly use in our program. Header files contain definitions of the funtions which can be incorporated into any C program by using pre-processor #include statement with the header file. Standard header files are provided with each complier, and covers a range of areas like string handling, mathematical funtions, data conversion, printing andreading of variables.

With time, you will have a clear picture of what header files are, as of now consider as a readymade piece of function which comes packaged with the C language and you can use them without worrying about how they work, all you have to do is include the header file in your program.

To use any of the standard functions, the appropriate header file must be included. This is done at the beginning of the C source file./

For example, to use the printf() function in a program, which is used to display anything on the screen, the line #include <stdio.h> is required because the header file **stdio.h** contains the printf() function. All header files will have an extension.h.

Example

❖ #include<stdio.h>

Definition Section

In this section, we define different constants. The keyword define is used in this part.

```
#define PI = 3.14
```

Global Declaration Section

This part of the code is the part where the global variables are declared. All the global variable used are declared in this part. The user-defined functions are also declared in this part of the code.

```
float area(float r);
int a = 7;
```

Main Function Section

Every C-programs needs to have the main function. Each main function contains 2 parts. A declaration part and an Execution part. The declaration part is the part where all the variables are declared. The execution part begins with the curly brackets and ends with the curly close bracket. Both the declaration and execution part are inside the curly braces.

OR

main() function is a function that must be there in every C program. Everything inside this function in a C program will be executed. In the above example, int written below the main() function is the **return type** of main() function. The curly braces {} just after the **main()** function encloses the **body of main()** function.

```
int main (void)
  {
int a = 10;
```



```
printf("%d", a);
return 0;
  }
```

Sub Program Section

All the user-defined functions are defined in this section of the program.

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

Sample Program

The C program here will find the area of a circle using a user-defined function and a variable pi holding the value of pi

```
* File Name: area of circle.c
* Author: Manthan Naik
* date: 09/08/20 19
* description: a program to calculate area of circle
* user enters the radius
*/
#include<stdio.h>          //link section
#define pi 3.14            //defination section
float area(float r);       //global declaration
int main()                 //main function
{
float r;
printf(" Enter the radius:r"):
scanf("%f',&r);
printf("the area is: %f'.area(r));
return 0:
}
float area(float r)
{
return pi * r * r;//sub program
}
```

Output :

```
C:\WINDOWS\SYSTEM\
Enter the RADIUS : 7
the area is: 153.860001
----------------------
(program exited with code: 0)
Press any key to continue - - -
```

Language Processors

Assembly language is machine dependent yet mnemonics that are being used to represent instructions in it are not directly understandable by machine and high Level language is machine independent. A computer understands instructions in machine code, *i.e.* in the form of 1's and 0's. It is a tedious task to write a computer program directly in machine code. The programs are written mostly in high level languages like Java, C++, Python etc. and are called source code. These source code cannot be executed directly by the computer and must be converted into machine language to be executed. Hence, a special translator system required.

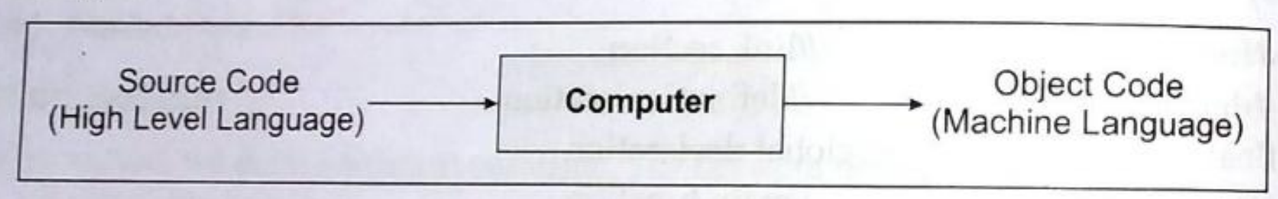
The language processors can be any of the following three types:

1. Compiler

The language processor that reads the complete source program written in high level language as a whole in one go and translates it into an equivalent program in machine language is called as a Compiler.

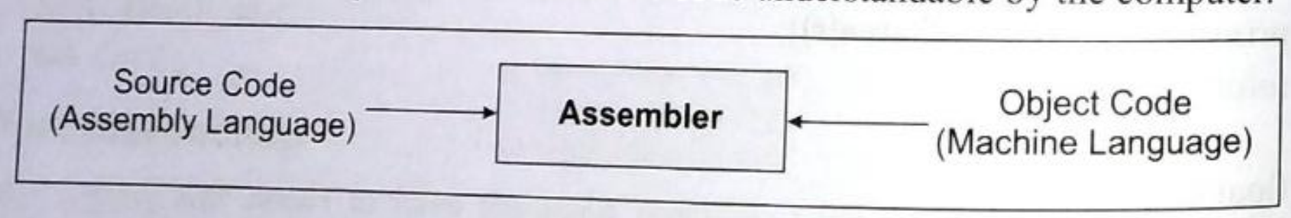
- **Example : C, C++, C#, Java**

In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again.>



2. Assembler

The Assembler is used to translate the program written in Assembly language into machine code. The source program is a input of assembler that contains assembly language instructions. The output generated by assembler is the object code or machine code understandable by the computer.



3. Interpreter

The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter. If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message. The interpreter moves on to the next line for execution only after removal of the error. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.

- **Example :** Perl, Python and Matlab.

Difference between Compiler and Interpreter

Compiler	Interpreter
Performs the translation of a program as a whole.	Performs statement by statement translation.
Execution is faster.	Execution is slower.
Requires more memory as linking is needed for the generated intermediate object code.	Memory usage is efficient as no intermediate object code is generated.
Debugging is hard as the error messages are generated after scanning the entire program only.	It stops translation when the first error is met. Hence, debugging is easy.
• **Examples :** C, C++, Java	• **Examples :** Python, Perl

❑

2 Program Structures

I/O statements in C Programming

There are some library functions which are available for transferring the information between the computer and the standard input and output devices.

These functions are related to the symbolic constant and are available in the header file.

Some of the input and output functions are as follows

(i) printf

This function is used for displaying the output on the screen *i.e.* the data is moved from the computer memory to the output device.

Syntax :

```
printf("format string", arg1, arg2,...);
```

In the above syntax, 'format string' will contain the information that is formatted. They are the characters which will be displayed as they are.

arg1, arg2 are the output data items.

Example : Demonstrating the printf function

```
printf("Enter a value:");
```

- Printf will generally examine from left to right of the string.
- The characters are displayed on the screen in the manner they are encountered until it comes across % of \.
- Once it comes across the conversion specifiers it will take the first argument and print it in the format given.

(ii) scanf

scanf is used when we enter data by using an input device

Syntax :

```
scanf("format string", &arg1, &arg2, .....);
```

The number of items which are successful are returned.

Format string consists of the conversion specifier. Arguments can be variables or array name and represent the address of the variable. Each variable must be preceded by an ampersand (&) Array names should never begin with an ampersand.

Example : Demonstrating scanf

```
int avg;
float per;
```



```
chargrade;
scanf("d%f%c", &avg, &per, &grade);
```

- scanf works totally opposite to printf. The input is read, interpret using the conversion specifier and stores it in the given variable.
- The conversion specifier for scanf is the same as **printf.**
- scanf reads the characters from the input as long as the characters match or it will terminate. The order of the characters that are entered are not important.
- It requires an enter key in order to accept an input.

(iii) getch

This function is used to input a single character. The character is read instantly and it does not require an enter key to be pressed. The character type is returned but it does not echo on the screen.

Syntax :

```
int getch(void);
ch=getch();
```

where,

ch-assigned the character that is returned by getch.

(iv) putch

this function is a counterpart of getch. Which means that it will display a single character on the screen. The character that is displayed is returned.

Syntax :

```
int putch(int):
putch(ch);
```

where,

ch -the character that is to be printed.

(iv) getche

This function is used to input a single character. The main difference between getch and getche is that getche displays the (echoes) the character that we type on the screen.

Syntax :

```
int getch(void);
ch=getche();
```

(vi) getchar

This function is used to input a single character. The enter key is pressed which is followed by the character-that is typed. The character that is entered is echoed.

Syntax :

```
ch-getchar
```

(vii) putchar

This function is the other side of getchar. A single character is displayed on the screen.

Syntax :

```
putchar(ch);
```

(viii) gets and puts

They help in transferring the strings between the computer and the standard input-output device. Only single arguments are accepted. The arguments must be such that it represents a string. It may include white space characters. If gets is used enter key has to be pressed for ending the string. The gets and puts function are used to offer simple alternatives of scanf and printf for reading and displaying.

Example :

```
#include <stdio.h>
void main()
{
char line[30];
  gets (line);
  puts (line);
}
```

As we all know the three essential functions of a computer are reading, processing and writing data. Majority of the programs take data as input, and then after processing the processed data is being displayed which is called information. In C programming you can use *scanf()* and *printf()* predefined function to read and print data.

Example :

```
#include <stdio.h>
void main()
{
int a, b, c;
printf("Please enter any two numbers: \n")
scanf ("%d%d.", &a, &b);
c = a + b;
printf("The addition of two number is : %d", c);
```

Output :

Please enter any two numbers:

12

3

The addition of two number is : 15

The above program *scanf()* is used to take input from the user, and respectively printf() is used to display output result on the screen.

Managing Input/Output

I/O operations are useful for a program to interact with users, *stdlib* is the standard C library for input-output operations. While dealing with input-output operations in C, two important streams play their role. These are :

1. Standard Input (stdin)
2. Standard Output (stdout)

Standard input or stdin is used for taking input from devices such as the keyboard as a data stream. Standard output or stdout is used for giving output to a device such as a monitor. For using I/O functionality, programmers must include *stdio header-file* within the program.

Reading Character

The easiest and simplest of all I/O operations are taking a character as input by reading that character from standard input (keyboard).getchar() function can be used to read a single character. This function is alternate to scanf() function.

Syntax :

```
varname = getchar();
```

Example :

```
#include <stdio.h>
void main()
{
char title;
title = getchar();
}
```

There is another function to do that task for files: getc which is used to accept a character from standard input.

Syntax :

```
int getc (FILE * stream);
```

Writing Character In C

Similar to qetchar() there is another function which is used to write characters, but one at a time.

Syntax :

```
putchar(varname);
```

Example :

```
#include <stdio.h>
void main()
{
char result = 'P';
putchnr(result);
putchar('\n');
}
```

Similarly, there is another function putc which is used for sending a single character to the standard output.

Syntax :

```
int putc (int c, FILE *stream);
```

Formatted Input

It refers to an input data which has been arranged in a specific format. This is possible in C using scanf(), We have already encountered this and familiar with this function

Syntax :

```
scanf("control string", arg1, arg2, ..., argn);
```

The field specification for reading integer inputted number is:

%w sd

Here the % sign denotes the conversion specification; *w* signifies the integer number that defines the field width of the number to be read, *d* defines the number to be read in integer format.

Example :

```
#include <stdio.h>
void main()
{
int var1 = 60;
int var2 = 1234;
scanf('%2d %5d", &var1, &var2);
}
```

Input data items should have to be.separated by spaces, tabs or new-line and the punctuation marks are not counted as separators.

Reading and Writing Strings

There are two popular library functions gets() and puts() Provides to deal with strings in C gets: The char *gets(char *str) reads a line, from stdin and keeps the string pointed to by the str and is terminated when the new line is read or *EOF* is reached. The declaration of gets() function is:

Syntax :

```
char *gets(char *str);
```

Where str is a pointer to an array of characters where C strings are stored.

puts; The function - int puts(const char *str) is used to write a string to stdout, but it does not include null characters. A new line character needs to be appended to the output. The declaration is :

Syntax :

```
int puts(const char *str)
```

where str is the string to be written in C.

Keywords

In c programming language there are set words that you cannot use as an identifier. These words are known as "reserved"words or "keywords". Keywords are standard identifiers and their meaning and purpose is predefined by the compiler.

List of all Keywords in C Language

There is set of 32 keywords in c programming language.

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

Description of all Keywords in C

Auto

The auto keyword declares automatic variables. For example :

```
auto int var1;
```

This statement suggests that var1 is a variable of storage class auto and type int.

Variables declared within function bodies are automatic by default. They are recreated each time a function is executed.

Since, automatic variables are local to a function, they are also called local variables.

Break and continue

The break statement makes program jump out of the innermost enclosing loop (while, do, for or switch statements) explicitly.

The continue statement skips the certain statements inside the loop.

```
for (i=1; i<=10; ++i)
{
  if (i==3)
  continue;
  if (i==7)
  break;
  printf("%d", i);
}
```

Output :

```
1 2 4 5 6
```

When I is equal to 3, continue statement comes into effect and skips 3. When I is equal to 7, break statement comes into effect and terminates the for loop.

Switch, case and default

The switch and case statement is used when a block of statements has to be executed among many blocks. For example :

```
switch(expression)
{
case 1:
//some statements to execute when 1
break;
case '5':
//some statements to execute when 5
break;
default:
//some statements to execute when default;
}
```

char

The char keyword declares a character variable. For example:

```
char alphabet;
```

Here, alphabet is a character type variable.

const

An identifier can be declared constant by using const keyword.

```
const int a = 5;
```

do...while

```
int i;
do
{
  printf("%d ", i);
  i++;
}
while (i<10)
```

double and float

Keywords double and float are used for declaring floating type variables. For example:

```
float number;
double long Number;
```

Here, number is single precision floating type variable whereas, long Number is a double precisi floating type variable.

if and else

In C programming, if and else are used to make decisions.

```
if (i == 1)
  printf("i is 1.")
else
  prinf("i is not 1.")
```

If value of i is other than 1, output will be :

i is not 1

enum

Enumeration types are declared in C programming using keyword enum. For example:

```
enum suit
{
  hearts;
  spades;
  clubs;
  diamonds;
};
```

Here, a enumerated variable suit is created having tags: hearts, spades, clubs and diamonds.

extern

The extern keyword declares that a variable or a function has external linkage outside of the fi is declared.

for

There are three types of loops in C programming. The for loop is written in C programming using keyword for. For example:

```
for (i=0; i< 9; ++i)
{
  printf("%d ",i);
}
```

Output :

0 1 2 3 4 5 6 7 8

goto

The goto keyword is used for unconditional jump to a labeled statement inside a function. For example:

```
for (i=l; i<5; ++i)
{
  if(i==10)
  goto error;
}
printf("i is not 10");
error:
  printf("Error, count cannot be 10.");
```

Output :

Error, count cannot be 10.

int

The int keyword declares integer type variable. For example:

```
int count;
```

Here, count is a integer variable.

short, long, signed and unsigned

The short, long, signed and unsigned keywords are type modifiers that alters the meaning of a base data type to yield a new type.

```
short intsmalllnteger;
long int biglnteger;
signed int normallnteger;
unsigned int positiveInteger;
```

Range of int type data types

Data types	Range
short int	– 32768 to 32767
long int	– 2147483648 to 214743648
signed int	– 32768 to 32767
unsigned int	0 to 65535

return

The return keyword terminates the function and returns the value.

```
int func()
{
  int b = 5;
  return b;
}
```

This function func() returns 5 to the calling function.

sizeof

The size of keyword evaluates the size of data (a variable or a constant).

```
#include<stdio.h>
int main()
{
  printf("%u bytes.",sizeof(char));
}
```

Output :

1 bytes.

register

The register keyword creates register variables which are much faster than normal variables.

```
register int var1;
```

static

The static keyword creates static variable. The value of the static variables persists until the end the program. For example:

```
static int var;
```

struct

The struct keyword is used for declaring a structure. A structure can hold variables of differe types under a single name.

```
struct student
{
  char name[80];
  float marks;
  int age;
}s1, s2;
```

typedef

The typedef keyword is used to explicitly associate a type with an identifier.

```
typedef float kg;
kg bear, tiger;
```

union

A Union is used for grouping different types of variable under a single name.


```
union student
{
  char name[80];
  float marks;
  int age;
}
```

void

The void keyword indicates that a function doesn't return any value.

```
void testFunction(int a)
{
.....
}
```

Here, function testFunction( ) cannot return a value because the return type is void.

volatile

The volatile keyword is used for creating volatile objects. A volatile object can be modified an unspecified way by the hardware.

```
const volatile number
```

Here, number is a volatile object.

Since, number is a constant variable, the program cannot change it. However, hardware can change it since it is a volatile object.

const

const can be used to declare constant variables. Constant variables are variables which, when initialized, can't change their value. Or in other words, the value assigned to them cannot be modified further down in the program.

Syntax :

```
const data_type var_name = var_value;
```

- **Note :** Constant variables must be initialized during their declaration, const keyword is also used with pointers. Please refer the const qualifier in C for understanding the same.

extern

extern simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can me made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

Syntax :

```
extern data_type var_name = var_value;
```

static

static keyword is used to declare static variables, which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere within that file as their scope is local to this file. By default, they are assigned the value 0 by the compiler.

Syntax :

```
static data_type var_name = var_value;
```

void

void is a special data type. But what makes it so special? void, as it literally means, is an empty data type. It means it has nothing or it holds no value. For example, when it is used as the return data type for a function it simply represents that the function returns no value. Similarly, when its added to a function heading, it represents that the function takes no arguments.

- **Note :** void also has a significant use with pointers. Please refer to the void pointer in C for understanding the same.

typedef

typedef is used to give a new name to an already existing or even a custom data type (like a structure). It comes in very handy at times, for example in a case when the name of the structure defined by you is very long or you just need a short-hand notation of a per-existing data type.

Let's implement the keywords which we have discussed above. Take a look at the following code which is a working example to demonstrate these keywords:

```
#include <stdio.h>
// declaring and initializing an extern variable
extern int x = 9;
// declaring and initializing a global variable
// simply int z; would have initialized z with
// the default value of a global variable which is 0
int z = 10;
// using typedef to give a short name to long long int
// very convenient to use now due to the short name
typedef long long int LL;
// function which prints square of a no. and which has void as its
// return data type
void calSquare(int arg)
{
  printf("The square of %d is %d\n", arg, arg *arg);
}
// Here void means function main takes no parameters
int main(void)
{
```



```
// declaring a constant variable, its value cannot be modified
const int a = 32;
//declaring a char variable
char b = 'G';
// telling the compiler that the variable z is an extern variable
//and has been defined elsewhere (above the main function)
extern int z;
LL c = 1000000;
printf("Hello World!\n");
// printing the above variables
printf("This is the value of the constant variable 'a': %d\n",a);
printf("'b' is a char variable. Its value is %c\n",b);
printf("'c' is a long long int variable. Its value is %lld\n",c);
printf("These are the values of the extern variables 'x' and 'z'"
" respectively: %d and %d\n",x,z);
//value of extern variable x modified
x = 2;
// value of extern variable z modified
z = 5;
// printing the modified values of extern variables 'x1 and 'z'
printf("These are the modified values of the extern variables"
"'x' and 'z' respectively: %d and %d\n",x,z);
// using a static variable
printf("The value of static variable 'y' is NOT initialized to 5 after the "
  "first iteration! See for yourself :)\n");
while (x > 0)
{
  static int y = 5;
  y++;
  // printing value at each iteration
  printf("The value of y is %d\n",y);
  x--;
}
//print square of 5
calSquare(S);
  printf("Bye! See you soon. :)\n");
  return 0;
}
```

Output :

```
Hello World
This is the value of the constant variable 'a': 32
'b' is a char variable. Its value is G
```

'c' is a long long int variable. Its value is 1000000
These are the values of the extern variables Y and 'z' respectively: 9 and 10
These are the modified values of the extern variables 'x' and 'z' respectively: 2 and 5
The value of static variable 'y' is NOT initialized to 5 after the first iteration! See for yourself :)
The value of y is 6
The value of y is 7
The square of 5 is 25
Bye! See you soon.:)

Basics usage of these keywords -

if, else, switch, case, default - Used for decision control programming structure.

break - Used with any loop OR switch case.

int, float, char, double, long– These are the data types and used during variable declaration.

for, while, do – types of loop structures in C.

void – One of the return type.

goto – Used for redirecting the flow of execution.

auto, signed, const, extern, register, unsigned – defines a variable.

return – This keyword is used for returning a value.

continue – It is generally used with for, while and dowhile loops, when compiler encounters this statement it performs the next iteration of the loop, skipping rest of the statements of current iteration.

enum – Set of constants.

sizeof – It is used to know the size.

struct, typedef - Both of these keywords used in structures (Grouping of data types in a single record).

union – It is a collection of variables, which shares the same memory location and memory storage.

volatile

Constant

Constants refers to the fixed values that do not change during the execution of a program. A "constant" is a number, character, or character string that can be used as a value in a program. Use constants to represent floating-point, integer, enumeration, or character values that cannot be modified.

There may be a situation in programming that the value of certain variables to remain constant during execution of a program. In doing so we can use a qualifier const at the time of initialization.

For example :

```
const float pie = 3.147;
const int radius = 4;
const char c = 'A';
const char namef[] = "Samina Kauser";
```

In C constant can also be used using preprocessor directive

For example :


```
#define FIRST_NUMBER1
```

Key point for constant :

- C Constants are also like normal variables. But, only difference is, their values can not be modified by the program once they are defined.
- Constants refer to fixed values. They are also called as literals
- Constants may be belonging to any of the data type.
- Syntax:

const data_type variable_name; (or) const data_type *variable_name;

Types of C constant

1. Integer constants
2. Real of Floating point constants
3. Octal & Hexadecimal constants
4. Character constants
5. String constants
6. Backslash character constants

Constant type	data type (Example)
Integer constants	int (53, 762, –478 etc.) unsigned into (5000u, 1000u etc) long int, long long int (473, 6472, 147, 483, 680)
Real or Floating point constants	float (10.456789) doule (600.123456789)
Octal constant	int (Example : 013/*starts with 0 * /)
Hexadecimal constant	int (Example: 0×90 /*starts with 0x*/)
Character constants	char (Example: 'A', 'B', 'C')
String constants	char (Example: "ABCD", "Hai")

Rules For Constructing C Constant

1. Integer Constants In C

The numbers with no fractional part are called integer constants. C accepts integer in three numbering systems that are decimal, octal and hexadecimal. When an integer has no prefix then C considers it as decimal integer, When it has 0 (zero) prefix then C considers it as Octal and when it has Ox (zero ex) as prefix then considers it as hexadecimal integer.

- An integer constant must have at least one digit.
- It must not have a decimal point
- it can either be positive or negative.
- No commas or blanks are allowed within an integer constant.
- If no sign precedes an integer constant,.it is assumed to be positive.
- The allowable range for integer constants is -32768 to 32767.

In C language printf() and scanf() are so intelligent so that they automatically convert from one system to another according to the formatting characters we use.

%0 to represent or convert to octal

%x or %X to represent or convert to hexadecimal.

%x display in small case
%X display in upper case
%d to represent or convert to decimal

Example :

```
#include<stdio.h>
int main()
{
int a = 10;
printf("%d\n%o\n%x\n%X",a,a,a,a);
return 0;
}
```

Output:

10
12
a
A

2. Real Constants in c

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.

Integer numbers are unable to represent distance, height, temperature, price, and so on. This information is containing fractional parts or real parts like 56.890. Such numbers are called Real or Floating Point Contents.

Example :

- A Real Constant must have at Least one Digit
- it must have a Decimal value
- it could be either positive or Negative
- if no sign is Specified then it should be treated as Positive
- No Spaces and Commas are allowed in Name

Following rules must be observed while constructing real constants expressed in fractional form :

(a) Real Constants must have a decimal point.
(b) A real constant must have at least one digit.
(c) Real Constants could be either positive or negative.
(d) Default sign is positive.
(e) No commas or blanks are allowed within a real constant.

3. Character And String Constants In C

A Character is Single Alphabet a single digit or a Single Symbol that is enclosed within Single inverted commas.

1. Character Constant **Can hold Single character at a time.**
2. Contains Single Character Closed within a pair of **Single Quote Marks**
3. Single Character is smallest **Character Data Type** in C.

4. **Integer Representation:** Character Constant reprent by **Unicode**
5. It is Possible to Perform Arithmetic Operations on Character Constants

Examples of Character Type :

'a'
'A'
'1'
'4343'
'#'
'-'
'<'
'X'
ETC.

How to Declare Character Variable :

Way 1: Declaring Single Variable

```
char variable_name;
```

Way 2: Declaring Multiple Variables

```
char var1,var2,var3;
```

Way 3: Declaring & Initializing

```
char var1 = 'A',var2,var3;
```

String Constant

Text enclosed in double quote characters (such as "example") is a string constant. It produces a block of storage whose type is array of char, and whose value is the sequence of characters between the double quotes, with a null character (ASCII code 0) automatically added at the end. All the escape sequences for character constants work here too. The compiler merges a series of adjacent string constants into a single string constant (before automatically adding a null character at the end). For example, "sample" "text" produces the same single block of storage as "sample text".

A string constant is a collection of characters, digits, special symbols and escape sequences that are enclosed in double quotations.

We define string constant in a single line as follows...

"This is btechsmartclass"

We can define string constant using multiple lines as follows...

"This
is
btechsmartclass "

We can also define string constant by separating it with white space as follows...

"This" "is" "btechsmartclass"

All the above three defines the same string constant.

Key point

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.
- The maximum length of a character constant is 1 character.

- String constants are enclosed within double quotes.

4. Backslash Character Constants In C

Certain ASCII characters are unprintable, which means they are not displayed on the screen or printer. Those characters perform other functions aside from displaying text. Examples are backspacing, moving to a newline, or ringing a bell.

They are used in output statements. Escape sequence usually consists of a backslash and a letter or a combination of digits. An escape sequence is considered as a single character but a valid character constant.

These are employed at the time of execution of the program. Execution characters set are always represented by a backslash (\) followed by a character. Note that each one of character constants represents one character, although they consist of two characters.

These characters combinations are called as **escape sequence.**

- There are some characters which have special meaning in C language.
- They should be preceded by backslash symbol to make use of special function of them.
- Given below is the list of special characters and their purpose

Backslash_character	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\\	Backslash
\v	Vertical tab
\a	Alert or bell
\?	Question mark
\N	Octal constant (N is an octal constant)
\XN	Hexadecimal constant (N–hex.. dcml cnst)

How To Use Constants in a C Program

We can define constant in a C program in the following ways.

1. By "const" keyword
2. By "#define" preprocessor directive

Please note that when you try to change constant values after defining in C program, it will through error.

1. Using the 'const' keyword

We create a constant of any datatype using 'const' keyword. To create a constant, we prefix the variable declaration with 'const' keyword.

The general syntax for creating constant using 'const' keyword is as follows...


```
const datatype constantName;
OR
const datatype constantName = value;
```

Example :

const int x = 10 ;

Here, 'x' is a integer constant with fixed value 10.

Example Program

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int i = 9;
  const int x = 10 ;
i = 15 ;
  x = 100;    //creates an error
  printf("i = %d\nx = %d", i, x ) ;
}
```

The above program gives an **error** because we are trying to change the constant variable value (x = 100).

Example Program Using Const Keyword in C

```
#include <stdio.h>
void rnain()
{
const int height = 100; /*int constant*/
const float number - 3.14; /*Real constant*/
const char letter = 'A'; /*char constant*/
const char letter_sequence[10] = "ABC"; /*string constant*/
const char backslash_char = '\?'; /*special char cnst*/
printf("value of height :%d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n", letter_sequence);
printf("value of backslash_char : %c \n", backslash_char);
}
```

Output :

```
value of height: 100
value of number: 3.140000
value of letter: A
value of letter_sequence : ABC
value of backslash char: ?
```

2. Using '#define' preprocessor:

We can also create constants using '#define' preprocessor directive. When we create constant using this preprocessor directive it must be defined at the beginning of the program (because all the preprocessor directives must be written before the global declaration).

We use the following syntax to create constant using '^define' preprocessor directive...

#define constantname value;

Example :

#define PI 3.14

Here, **PI** is a constant with value **3.14**

Example Program :

```
#include<stdio.h>
#include<conio.h>
#defien PI 3.14
void main()
{
  int r, area ;
  printf("Please enter the radius of circle : ") ;
  scanf("%d", &r) ;
  area = PI * (r * r) ;
  priritf("Area of the circle = %d", area) ;
}
```

Example Program Using #Define prepprocessor Directive in C :

```
#include <stdio.h>
#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char'\?'
void main()
{
printf("value of height : %d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n",letter_sequence);
printf("value of backslash_char : %c \n",backslash_char);
}
```

Output :

```
value of height: 100
value of number: 3.140000
value of letter: A
value of letter_sequence : ABC
value of backslash_char: ?
```


Variables

Variables in C language plays an important role. We can also say that variables are the backbone of many programming languages. Variables in C languages are used to store different forms of data. It acts as a memory card where it saves all the data and used it during program execution. There are different types of variables in C, according to their types, the amount of memory or storage space it requires differs. As we said, variables in C are storage used to hold the value. Data that variables can be different like int, float, char, double, etc. All the code or program depends on the variables as it describes the type of data for execution.

"a variable is a container (storage area) to hold data."

To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

```
int playerScore = 95;
```

Here, playerScore is a variable of int type. Here, the variable is assigned an integer value 95. The value of a variable can be changed, hence the name variable.

```
char ch = 'a';
//some code
ch = 'I';
```

Rules for naming a variable

- Variables in C must not start with the number else the Variable will not be valid. For example (1 string, is not a valid variable).
- Blank space between variables is not allowed. For example, (string one is not valid, string_one is a valid variable).
- Keywords are not allowed to define as a variable. For example, (for is not a valid variable as it is used as a keyword in C language).
- As C is a case sensitive language, upper and lower cases are considered as a different variable. For Example (NUMBER and number will be treated as two different variables in C).
- Variable names can be a combination of string, digits, and special characters like underscores (_).

How to Work

- While declaring variables it tells compilers the type of data it holds.
- Variables tell compilers the name of the variables that are being used in the program.
- As variables specify storage, compilers do not have to worry about the memory location fo the variables, until they are declared.

How to Declare

Variables should be declared first before the program as it plays an important role. The syntax fo variables declaration is as follows :

data_type variable_name;

where,

- **data_type:** Indicates types of data it stores. Data types can be int, float, char, double, long i etc.

- **variable_name:** Indicates the name of the variable. It can be anything other than th keyword.

- **Note :** Declaration of variables must be done_before they are usedjn the program. Declaration does the following things.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. Until the variable is defined the compiler doesn't have to worry about allocating memory spac to the variable.
4. Declaration is more like informing the compiler that there exist a variable with followin datatype which is used in the program.
5. A variable is declared using the extern keyword, outside the mainQ function.

For example :

1. int a;
2. int a, b, c;

In the first example, int is a data type and a is a variable name. In the second example, we hav declared three variables a, b, and c.

After variables are declared, the space for those variables has been assigned as it will be used fo the program.

How to Initialize

Initialization of a variable is of two types:

- **Static Initialization:** Here, the variable is assigned a value in advance during writing program. This variable then acts as a constant.
- **Dynamic Initialization:** Here, the variable is assigned a value at the run time. The value o this variable can be altered every time the program is being run.

Different ways of initializing a variable in C

Method 1 (Declaring the variable and then initializing it)

```
int a;
a = 5;
```

Method 2 (Declaring and Initializing the variable together):

```
int a = 5;
```

Method 3 (Declaring multiple variables simultaneously and then initializing them separately)

```
int a, b;
a = 5;
b = 10;
```

Method 4 (Declaring multiple variables simultaneously and then initializing them simultaneously)

```
int a, b;
a = b = 10;
int a, b = 10, c = 20;
```

Method 5 (Dynamic Initialization : Value is being assigned to variable at run time.)

```
int a;
printf("Enterthe value of a");
scanf("%d", &a);
```


Types of Variables

There are 5 types of variables which are as follows:

1. Local variables
2. Global variables
3. Static variables
4. Automatic variables
5. External variables

Local Variable

Declaring a variable inside the function or block is called **local declaration.** The variable declared using local declaration is called **local variable.** The local variable can be accessed only by the function or block in which it is declared. That means the local variable can be accessed only inside the function or block in which it is declared.

Let's take an example.

```
tfinclude <stdio.h>
int main(void)
{
   for (int i = 0; i < 5; ++i)
   {
   printf("C programming");
   }
   // Error: i is not declared at this point
    printf("%d", i);
    return 0;
}
```

When you run the above program, you will get an error undeclared identifier i. It's because i is declared inside the for loop block. Outside of the block, it's undeclared.

Let's take another example.

```
int main()
{
   int nl; // nl is a local variable to main()
}
void func()
   {
   int n2; // n2 is a local variable to func()
   }
```

In the above example, nl is local to main() and n2 is local to func().

This means you cannot access the nl variable inside func() as it only exists inside main().

Similarly, you cannot access the n2 variable inside main() as it only exists inside func().

Global Variable

Declaring a variable before the function definition (outside the function definition) is called **global declaration**. The variable declared using global declaration is called global variable.

The global variable can be accessed by all the functions that are defined after the global declaration. That means the global variable can be accessed any where in the program after its declaration.

Example 1 : Global Variable

```
#include <stdio.h>
void display();
int n = 5; //global variable
int main()
{
  ++n;
  display();
  return 0;
}
void display()
{
  ++n;
  printf("n = %d", n);
}
```

Output :

```
n = 7
```

Static variable

Static variables are initialized only once. The compiler persists with the variable till the end of the program. Static variables can be defined inside or outside the function. They are local to the block. The default value of static variables is zero. The static variables are alive till the execution of the program.

Here is the syntax of static variables in C language,

```
static datatype variable_name = value;
```

Here,

datatype – The datatype of variable like int, char, float etc.

variable_name – This is the name of variable given by user.

value – Any value to initialize the variable. By default, it is zero.

Here is an example of static variable in C language,

Example :

```
#include <stdio.h>
int main()
{
  auto int a = -28;
  static int b = 8;
  printf("The value of auto variable : %d\n", a);
  printf("The value of static variable b : %d\n",b);
  if (a ! = 0)
  printf("The sum of static variable and auto variable : %d\n",(b+a));
  return 0;
}
```

Output :

Here is the output


```
The value of auto variable: -28
The value of static variable b: 8
The sum of static variable and auto variable: -20
```

Difference between Variable and Identifier

An Identifier is a name given to any variable, function, structure, pointer or any other entity in a programming language. While a variable, as we have just learned in this tutorial is a named memory location to store data which is used in the program.

Identifier	Variable
Identifier is the name given to a variable, function etc.	While, variable is used to name a memory location which stores data.
An identifier can be a variable, but not all identifiers are variables.	All variable names are identifiers.
Example : //a variable int studypoint; //or, function int studypoint() { .... }	**Example :** //int variable int x; //float variable float y;

Data Type

A programming language is proposed to help programmer to process certain kinds of data and to provide useful output. The task of data processing is accomplished by executing series of commands called program. A program usually contains different types of data types (integer, float, character etc.) and need to store the values being used in the program. C language is rich of data types. A C programmer has to employ proper data type as per his requirements

C has different data types for different types of data and can be broadly classified as :

There are three classes of Data-Type

- Primary Data Type
- Derived Data Type
- User Defined Data Type

Primary Data Type

The C programming language supports primary data types. A primary type is predefined by the language and is named by a reserved keyword.

Integer data type :

- Integer data type allows a variable to store numeric values.
- "int" keyword is used to refer integer data type.
- The storage size of int data type is 2 or 4 or 8 byte.
- It varies depend upon the processor in the CPU that we use. If we are using 16 bit processor, 2 byte (16 bit) of memory will be allocated for int data type.

- Like wise, 4 byte (32 bit) of memory for 32 bit processor and 8 byte (64 bit) of memory for bit processor is allocated for int datatype.
- int (2 byte) can store values from -32,768 to +32,767
- int (4 byte) can store values from -2,147,483,648 to +2,147,483,647.
- If you want to use the integer value that crosses the above limit, you can go for "long int" a "long long int" for which the limits are very high.

Note :

- We can't store decimal values using int data type.
- If we use int data type to store decimal values decimal values will be truncated and we will only whole number.
- In this case, float data type can be used to store decimal values in a variable.

Character data type:

- Character data type allows a variable to store only one character.
- Storage size of character data type is 1. We can store only one character using character d type.
- "char" keyword is used to refer character data type.
- For example, 'A' can be stored using char datatype. You can't store more than one charac using char data type.
- Please refer C - Strings topic to know how to store more than one characters in a variable

Floating point data type

Floating point data type consists of 2 types. They are,

1. float
2. double

1. float :

- Float data type allows a variable to store decimal values.
- Storage size of float data type is 4. This also varies depend upon the processor in the CPU "int" data type.
- We can use up-to 6 digits after decimal using float data type.
- For example, 10.456789 can be stored in a variable using float data type.

2. double :

- Double data type is also same as float data type which allows up-to 10 digits after decim
- The range for double datatype is from 1E-37 to 1E+37.

void data Type : void(used for function when no value is to be return)

Derived Data Type

These data types are defined by user itself. Like a collection of integer stored contiguous makes array. So derived data types are what we can create by combining various primitive data types. Usi derived data types, an infinite variety of new types can be formed. **The array and structure types collectively called the aggregate types.** Note that the aggregate types do not include union types, a union may contain an aggregate member.

Types of derived data types in C

There are basically five derived types in C :

- Array types
- Structure types
- Union types
- Function types
- Pointer types
- Enumeration

1. Array

Arrays are one of the mostly used derived data types in C and they can be formed by collecting the primitive data types like – float , int or char. So when you make the collection of any of these data types then it forms an array. And they are stored in contiguous memory location.

An array type can be formed from any valid completed type. Completion of an array type requires that the number and type of array members be explicitly or implicitly specified. The member types can be completed in the same or a different compilation unit.

Typically, arrays are used to perform operations on some homogeneous set of values. The size of the array type is determined by the data type of the array and the number of elements in the array. Each element in an array has the same type. For example, the following definition creates an array of four characters:

```
char x[] = "HEY" /* Declaring an array x */;
```

Each of the elements has the size of a char object, 8 bits. The size of the array is determined by its initialization; in the previous example, the array has three explicit elements plus one null character. **Four elements of 8 bits each results in an array with a size of 32 bits or we can simply say 4 byte.**

An array is allocated contiguously in memory, and cannot be empty (that is, have no members). An array can have only one dimension. To create an array of "two dimensions," declare an array of arrays, and so on.

It is possible to declare an array of unknown size; this sort of declaration is called an *incomplete array declaration*, because the size is not specified. The following example shows an incomplete declaration:

```
int x[];
```

The size of an array declared in this manner must be specified elsewhere in the program.

2. Structures

Structures are also one of the derived data types in C and they can form the collection of dissimilar data types like an int, a float and array of char. So whenever you require to put dissimilar data type together you can use a structure.

So a structure type is a sequentially allocated nonempty set of objects, called members. Structures let you group heterogeneous data. Unlike arrays, the elements of a structure need not be of the same data type. Also, elements of a structure are accessed by name, not by subscript. The following example declares a structure employee, with two structure variables (e1 and e2) of the structure type employee :

```
struct employee
{
char name[30];
int age;
```

```
    int empnumber;
    };
    struct employee e1, e2;
```

Structure members can have any type except an incomplete type, such as the void type or function type. **Structures can contain pointers to objects of their own type, but they cannot contain an object of their own type as a member;** such an **object would have an incomplete type.** For example:

```
    struct employee
    {
    char name[30];
    struct employee point; /* This is invalid. */
    int *f();
    };
```

The following example, however, is valid:

```
    struct employee
    {
    char name[30];
    struct employee *point;/*  Member can contain pointer to employee structure. */
    int (*f)(); /*  Pointer to a function returning int*/
    };
```

The name of a declared structure member must be unique within the structure, but it can be used in another nested or unnested structure or name spaces to refer to a different object. For example:

```
    struct
    {
    int x;
    struct
      {
      int x;  /* This 'x' refers to a different object than the previous 'x'*/
      };
    };
```

The compiler assigns storage for structure members in the order of member declaration, with increasing memory addresses for subsequent members. The first member always begins at the starting address of the structure itself. **Subsequent members are aligned per the alignment unit, which may differ depending on the member sizes in the structure. A structure may contain padding (unused bits) so that members of an array of such structures are properly aligned, and the size of the structure is the amount of storage necessary for all members plus any padded space needed to meet alignment requirements.**

3. Union

Union is also a derived data type in C and they are much like structure but the amount of memory used by them is equivalent to the size of union member which took maximum storage space .

A union type can store objects of different types at the same location in memory. The different union members can occupy the same location at different times in the program. The declaration of a union includes all members of the union, and lists the possible object types the union can hold. The union can hold any one member at a time-subsequent assignments of other members to the union overwrite the existing object in the same storage area.

Unions can be named with any valid identifier. An empty union cannot be declared, nor can a union contain an instance of itself. **A member of a union cannot have a void , function, or incomplete type.** Unions can contain pointers to unions of their type.

Another way to look at a union is as a single object that can represent objects of different types at different times. Unions let you use objects whose type and size can change as the program progresses, without using machine-dependent constructions. Some other languages call this concept a *variant record*.

The syntax for defining unions is very similar to that for structures. Each union type definition creates a unique type. Names of union members must be unique within the union, but they can be duplicated in other nested or unnested unions or name spaces. For example:

```
    union
    {
    int x;
    union
      {
      int x;  /* This 'x' refers to a different object than the previous 'x'*/
      };
    };
```

The size of a union is the amount of storage necessary for its largest member, plus any padding needed to meet alignment requirements.

Like if you have a union of an array of 5 integer , a float and a array of 5 char then – the union storage space would be equal to 20 bytes for 64 bit compiler.

4. Function

A function type describes a function that returns a value of a specified type. If the function returns no value, it should be declared as "function returning void " as follows:

```
    void function ();
```

In the following example, the data type for the function is "function returning int":

```
    int add()
    {
    int a=10,b=10,c;
    c=a+b;
    return c;
    }
```

How to calculate size of a function?

Size of any function is calculated as:

Size of function = Size of all local variable which has declared in function + Size of those global variables which has used in function+ Size of all its parameter+ Size of returned value if it is an address. Example:

What is size of add function?

```
int add()
{
int a=10,b=10,c;
c=a+b;
return c;
}
```

so the answer is it's 12 byte for 64 bit compiler.

5. Pointer

Pointer are fourth type of derived data types in C though the size that they take in memory is always fix but the type of pointer depends on the type of element whose address they store.

Pointers are considered by many to be complex in C, but that is not the case. Simply put, a pointer is just a variable that stores the address of another variable. A pointer can store the address of variables of any data types. This allows for dynamic memory allocation in C. Pointers also help in passing variables by reference.

The pointer is defined by using a '*'operator. For example –

```
int *ptr;
```

This indicates ptr stores an address and not a value. To get the address of the variable, we use the dereference operator '&.' The size of a pointer is 2 bytes. Pointers cannot be added, multiplied, or divided. However, we can subtract them. This will help us know the number of elements present between the two subtracted pointers.

Example,

```
#include <stdio.h>
int main()
{
 int num = 10;
 printf("Value of variable num is: %d", num);
 printf("\nAddress of variable num is: %p", &num);
 return 0;
}
```

So it's like if they store address of an integer then there type would be integer pointer and if they store the address of an array then there type would be array pointer.

Enumeration:

Enumeration is a special data type that consists of integral constants, and each of them is assigned with a specific name. "enum" keyword is used to define the enumerated data type.

The most common example of this is the days of the week.

```
enum weekdays;
enum weekend;
```

The enum keyword is also used to define the variables of enum type. There are two ways to define the variables of enum type as follows.

```
enum week{sunday, monday, tuesday, wednesday, thursday, friday, saturday};
```



```
enum week day;
#include<stdio.h>
enum week{Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun};
enum day{Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund};
int main()
{
  printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",
  Mon , Tue, Wed, Thur, Fri, Sat, Sun);
  printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",
  Mond, Tues, Wedn, Thurs, Frid, Satu, Sund);
  return 0;
}
```

Output

```
The value of enum week: 10111213101617
The default value of enum day: 0123181112
```

In the above program, two enums are declared as week and day outside the main() function. In the main() function, the values of enum elements are printed.

```
enum week{Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun};
enum day{Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund};
int main()
{
printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",
Mon , Tue, Wed, Thur, Fri, Sat, Sun);
printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",
Mond , Tues, Wedn, Thurs, Frid, Satu, Sund);
}
```

User Define Data Type

By using a feature known as "type definition" that allows user to define an identifier that would represent a data type using an existing data type.

Note :

General form :

```
typedef type identifier;
or (to better understand)
typedef existing_data_type new_user_define_data_type;
Example Using user defined data type
typedef int number;
typedef long big_ number;
typedef float decimal;
typedef double big_decimal;
/********* now we can use above user defined types to declare variables********/
```

```
number visitors = 25;
big_number population = 12500000;
decimal radius = 3.5;
big_decimal pie= 3.1415926535
```

Advantage

Main advantage of typedef is that we can create meaningful data type names for increasing the readability of the program.

Operator

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. **Operators** are used to connect **operands**, *i.e.* constants and variables, to form expressions.

For example, in the expression a+b, a and b are operands and + is the operator. Depending on the number of operands on which an operator operates, the operators in C language can be grouped into three categories: unary operators, binary operators and ternary operators. A unary operator takes only one operand, as in -x, whereas a binary operator operates on two operands, as in a+b. A ternary operator takes three operands, as in (a > b) ? a : b, where the conditional expression operator (? :) is a ternary operator.

C language is rich in built-in operators and provides the following types of operators :

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

I. Arithmetic operators

Arithmetic operators are used to perform arithmetic operations on arithmetic operands, e., operands of integral as well as floating type. There are five arithmetic operators, +, –, • , /, and %, which respectively represent the processes of addition, subtraction multiplication, division, and modulus. The modulus operator (%) gives the remainder when one integer is divided by another integer. All of the five operators have been described with examples of codes in Table.

Arthmetic Operators

Operation	C operator	Algebraic expression	Exampls of code
Addition	+	x + y	int x, y, sum; sum = x + y;
Subtraction	–	x – y	float x, y, z; z = x – y,
Multiplication	×	x × y or x y	int a, b, c; a = b × c;
Divison	/	A % B or A/B	double A, B, x; x = A/B;
Modulus	%	In gives the remainder when an integer is divided by another integer.	int X, Y, M, M =X/Y; X = 12; Y = 5; M = 2

It should be noted that each of these operators has a precedence level, that is, priority in operation. If an expression contains operation of multiplication and addition, the -multiplication is performed first because it has higher precedence level than that of thd[addition. Table shows the precedence level of different operators. In many arithmetic* operations, the parentheses () are also used. It is generally employed to change the precedence level, that is, priority in operation. The precedence determines when a particular operator will be implemented when there are several operators in a single formula.

Comparative Priority of Arithmetic Operators

Operator	Priority
()	First If nested. the inner most is first
*, /, and %	Next toil If several from left to right
+ , -	Next to *, /, %. If several, from left to right.

```
#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
```

2. Assignment operators:

We have used the assignment operator (=), which is often called equal to in algebra. On the left of this operator we write the name of variable or l-value to which a value is to be assigned, and on right side we write the value to be assigned to it or r-value. The l-value is the memory segment in which the r-value is stored. Let A, B, and ch is the names of three variables declared as given below.

```
int A; // declaration allocates a memory space for A
A = 5; // assignment- stores value 5 in the memory
float B; // declaration allocates memory for B
B = 12.6; // store value 12.6 in the memory
char ch; // declaration allocates memory for ch
```

ch = 'H'; // Puts 'H' in the memory

We may as well combine the declaration and the assignment illustrated as follows.

```
int A =5;
float B -12.6;
char ch = 'H';
```

In the assignment of ch, note that the character value 'H' is enclosed between single quotes (''). Whenever the value is a character, it has to be enclosed between single quotes. On the other hand, if the value is a string of characters, it is enclosed in double quotes as illustrated below.

```
char Name[10] ="Dinesh";
```

Here, Name is a string, i.e., it is an array of characters. Because of this reason we have used the array symbol [ ] with number 10 after Name. Number 10 indicates the number of characters in the string including the null character ('\0') which marks the end of the string. The null character is automatically appended by the system whenever the value assigned is enclosed in double quotes.

3. Relational operators

The C language provides four relational and two equality operators for comparing the values of expressions. The relational operators are less than (<), greater than (>), less than or equal to (<=) and greater than or equal to (>-). The equality operators are equal to (==) and not equal to (/=). These operators are binary infix operators, i. e., they are used in the form a op b, where a and bare operands (constants, variables or expressions) and op is a relational or an equality operator. Table summarizes these operators.

Note that the token for equality operator is = A beginner often makes the mistake of writing the equal to operator simply as -, which is an assignment operator.

Thus, a == b means the values of variables a and b are compared for equality, whereas a = b means the value of variable b is assigned to variable a.

Table Relational and equality operators in the C language

Category	Operator	Meaning	Example	Associativity
Relation	<	Less than	a < b	Let to right
	>	Greater than	a > b	
	<=	Less than or equal to	a <= b	
	>=	Greater than or equal to	a >= b	
Equality	==	Equal to	a == b	
	!=	Not equal to	a! = b	

4. Logical operators

The C language provides three **logical operators** that can be used to join relational an< equality expressions and form complex **Boolean expressions,** *i.e.*, expressions with operands having true or false values. These operators include logical AND (&&), logical OR (\ \) and logical NOT (!). They are summarized in Table.

Note that the logical not (!) is a unary prefix operator. It is used, as in ! expr, where expr is a relational expression. The other two operators are binary infix operators. They are used, as in exprl op expr2, where exprl and expr2 are relational expressions.

Logical operators in the C language

Operator	Meaning	Example	Associativity
j	Logical not	! (a < 0)	Right to left
&&	Logical and	(a >= 0) && (a <= 100)	Left to right
II	Logical or	(a < 0) 1 1 (a > 100)	

Logical AND Operator : As the **logical AND(&&)** operator is a binary infix operator, it is used, as in exprl && expr2 where exprl and expr2 are relational expressions. This expression evaluates as true, *i.e.*, 1 if both exprl and expr2 are true; otherwise, it evaluates as false.

Consider the expression

```
(a >= 0) && (a <= 100)
```

This expression evaluates as true if both the relational expressions (a >= 0) and (a <= 100) are true, i.e., if the value of variable a is in the range 0 to 100 (both inclusive); otherwise, it evaluates as false, i. e., 0.

Logical OR Operator : The logical or (11) operator is also a binary infix operator. It is used, as in expr1 || expr2

where exprl and expr2 are relational expressions. This expression evaluates as true, *i.e.*, 1 if either exprl or expr2 is true (or both of them are true); otherwise, it evaluates as false.

Consider the expression

```
(a < 0) 11 (a > 100)
```

This expression evaluates as true if the value of variable a is less than zero or greater than 100, *i.e.*, outside the range 0 to 100; otherwise, it evaluates as false, *i. e.*, 0.

Logical NOT Operator : The **logical not (!)** operator performs logical complement operation. This is a unary prefix operator and is used, as in

```
! expr
```

This expression evaluates as true if expression expr is false, and false otherwise. Consider the expression

```
! (a < 0)
```

If the value of variable a is less than zero, this expression evaluates as false; otherwise, it evaluates as true. Not that this expression is equivalent to the relational expression a > = 0.

Consider another expression given below.

```
! ((a < 0) || (a > 100))
```

If the value of variable a is outside 0 to 100, the expression (a < 0: 11 (a > 100) evaluates as true and the given expression evaluates as false. On the other hand, if the value of a is in the range 0 ... 100, the above expression evaluates as true. Thus, the above expression is equivalent to the expression (a >= 0) & & (a <= 100).

5. Bit wise operators

All the objects that are stored in a computer are ultimately converted into binary numbers which are sequences of O's and 1's. Each digit in a binary number is stored on one bit of the computer memory. A bit is defined as the smallest unit of memory in a computer. In fact, computer manipulates a number by manipulating the bits on which the number is stored. In control systems also we often need to use operators to manipulate bits.

The C language provides six **bitwise operators** to manipulate the bit patterns of integral values (integers and characters). They include not, *i.e.,* **complement(~)**, and (&), **or** (|), **exclusive or,** *i.e.*, **xor (^), left shift (<<)** and **right shift (>>)**. These operators work directly on the bit patterns of the operands, *i.e.*, on sequences of bits (0 and 1) representing the operands. In addition, C language provides five compound assignment operators (&=, | =, ^=, <<= and >>=).

The **complement** operator (~) is a unary prefix operator and is used, as in ~a, whereas all other operators are binary infix operators and are used as in a op b.

First consider these bitwise operations on individual bits. The **bitwise and** operator evaluates as 1 if both operands are 1, and zero otherwise. The **bitwise** or operator evaluates as 1 if either or both operands is 1, and zero otherwise. The **bitwise xor** operator evaluates as 1 if either of the operands (but not both) is 1, and zero otherwise. Finally, the **bitwise** not operator complements the value of the operand, *i.e.*, it returns 1 if the operand is zero and vice versa.

6. Conditional operators {ternary operators)

The **conditional expression operator** (? :) is the only ternary operator in the C language. The conditional selection operator(? :) is more convenient to use than if-else provided there are only two options to choose from. It takes three operands and is used to evaluate one of the two alternative expressions depending on the outcome of a test expression as shown below.

test_*expr ? exprl: expr2*.

Here, test_expr is followed by a question mark(?), is the test express and expr1 and expr2 separated by a colon (:) are the alternative expressions, only one t which is evaluated, depending on the outcome oftest_expr. Observe how the questio mark (?) and the colon (:) separate these expressions. If test_ expr evaluates as true (i. e., non-zero), the value of the entire expression is equal to the value of expr 1; otherwise, it is equal to the value of expr2. Consider, for example, the expression a< b? a : b. The value of this expression is either a or b, depending on whether the expression is true or false, respectively. Thus, the expression determines the smaller of the two numbers a and b.

7. 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 —;

++ and ~ operator as prefix and postfix :

- If you use the ++ operator as prefix like: ++var. The value of var is incremented by 1 then, it returns the value.
- If you use the ++ operator as postfix like: var++. The original value of var is returned first then, var is incremented by 1.

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".

```
//Example for increment operators
#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".

```
//Example for decrement operators
#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

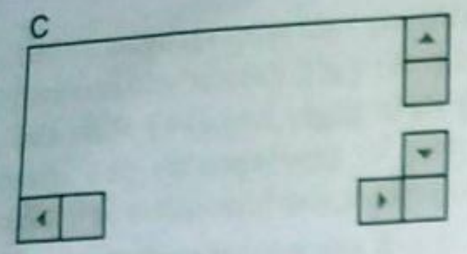
8. Special operators

Below are some of the special operators that the C programming language offers.

Operators	Description
&	This is used as pointer to a variable. Example : & a will give address of a.
*	This is used as pointer to a variable. **Example :** *a where, *is pointer to the variable a.
Sizeof()	This gives the size of 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;
int result, var1=10, var2=3;
result=var1/var2;
```

In this case, after the division performed on variables var1 and var2 the result stored in the variable "result" will be in an integer format. Whenever this happens, the value stored in the variable "result" loses its meaning because it does not consider the fraction part which is normally obtained in the division of two numbers.

Typecasting

Typecasting is converting one data type into another one. It is also called as data conversion or type conversion. It is one of the important concepts introduced in 'C' programming.

'C' programming provides two types of type casting operations:

1. Implicit type casting

2. Explicit type casting

Implicit Type Casting

Implicit type casting means conversion of data types without losing its original meaning. This type of typecasting is essential when you want to change data types **without** changing the significance of the values stored inside the variable.

Implicit type conversion happens automatically when a value is copied to its compatible data type. During conversion, strict rules for type conversion are applied. If the operands are of two different data types, then an operand having lower data type is automatically converted into a higher data type. This type of type conversion can be seen in the following example.

```
#include<stdio.h>
int main()
{
  short a=10; //initializing variable of short data type
  int b; //declaring int variable
  b=a; //implicit type casting
  printf("%d\n",a);
  printf("%d\n",b);
}
```

Output

10

10

```
#include<stdio.h>
int main()
{
  1 short a=10; //initializing variable of short data type
    int b; 2 // declaring int variable
  3 b=a; //implicit type casting
    printf("%d\n", a);
    printf("%d\n", b);
}
```

1. In the given example, we have declared a variable of short data type with value initialized as 10.
2. On the second line, we have declared a variable of an int data type.
3. On the third line, we have assigned the value of variable s to the variable a. On third line implicit type conversion is performed as the value from variable s which is of short data type is copied into the variable a ,which is of an int data type.

Converting Character to int

Consider the example of adding a character decoded in ASCII with an integer:

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

```
{
  int number = 1;
  char character = 'k'; /*ASCII value is 107 */
  int sum;
  sum = number + character;
  printf("Value of sum : %d\n", sum );
}
```

Output:

Value of sum : 108

Here, compiler has done an integer promotion by converting the value of 'k' to ASCII before performing the actual addition operation.

Consider the following example to understand the concept:

```
#include <stdio.h>
main()
{
  int num = 13;
  char c = 'k'; /* ASCII value is 107 */
  float sum;
  sum = num + c;
  printf("sum = %f\n", sum );
}
```

Output:

sum = 120.000000

First of all, the c variable gets converted to integer, but the compiler converts **num** and **c** into "float" and adds them to produce a 'float' result.

Important Points about Implicit Conversions

- Implicit type of type conversion is also called as standard type conversion. We do not require any keyword or special statements in implicit type casting.
- Converting from smaller data type into larger data type is also called as **type promotion.** In the above example, we can also say that the value of s is promoted to type integer.
- The implicit type conversion always happens with the compatible data types.

We cannot perform implicit type casting on the data types which are not compatible with each other such as:

1. Converting float to an int will truncate the fraction part hence losing the meaning of the value.
2. Converting double to float will round up the digits.
3. Converting long int to int will cause dropping of excess high order bits.

Explicit Type Casting

In implicit type conversion, the data type is converted automatically. There are some scenarios in which we may have to force type conversion. Suppose we have a variable div that stores the division of two operands which are declared as an int data type.

```
int result, var1=10, var2=3;
result=var1/var2;
```


In this case, after the division performed on variables var1 and var2 the result stored in the variable "result" will be in an integer format. Whenever this happens, the value stored in the variable "result" loses its meaning because it does not consider the fraction part which is normally obtained in the division of two numbers.

To force the type conversion in such situations, we use explicit type casting.

It requires a type casting operator. The general syntax for type casting operations is as follows:

```
(typename) expression
```

Here,

- The typename is the standard 'C' language data type.
- An expression can be a constant, a variable or an actual expression.

Let us write a program to demonstrate implementation of explicit type-casting in 'C'.

```
#include<stdio.h>
int main()
{
  float a = 1.2;
  //int b  = a; //Compiler will throw an error for this
  int b = (int)a + 1;
  printf("Value of a is %f\n", a);
  printf("Value of b is %d\n",b);
  return 0;
}
```

Output:

Value of a is 1.200000

Value of b is 2

```
#include<stdio.h>
int main()
{
    float a = 1.2; (1)
(2) //int b = a; // Compiler will throw a
    int b = (int) a + 1; (3)
(4) printf("Value of a is %f\n", a);
    printf("Value of b is %d/n", b) (5)
    return 0;
}
```

1. We have initialized a variable 'a' of type float.
2. Next, we have another variable 'b' of integer data type. Since the variable 'a' and 'b' are of different data types, 'C' won't allow the use of such expression and it will raise an error. In some versions of 'C,' the expression will be evaluated but the result will not be desired.
3. To avoid such situations, we have typecast the variable 'a' of type float. By using explicit type casting methods, we have successfully converted float into data type integer.
4. We have printed value of 'a' which is still a float

5. After typecasting, the result will always be an integer 'b.'

Formatted Output

The function printf() is used for formatted output to standard output based on a form specification. The format specification string, along with the data to be output, are the parameters t the printf() function.

Syntax :

printf (format, data1, data2,.....);

In this syntax format is the format specification string. This string contains, for each variable to b output, a specification beginning with the symbol % followed by a character called the conversio character.

Example :

printf ("%c", data1);

The character specified after % is called a conversion character because it allows one data type t be converted to another type and printed.

See the following table conversion character and their meaning

Conversion Character	Meaning
d	The data is converted to decimal (integer)
c	The data is taken as a character.
s	The data is a string and character from the until a NULL, character is reached.
f	The data is output as float or double with a default Precision 6.

Symbol	Meaning
\n	For new line (linefeed return)
\t	For tab space (equivalent of 8 spaces)

Example :

```
printf ("%c\n",data1);
```

The format specification string may also have text.

Example :

```
printf ("Character is:"%c\n". data1):
  z = a – (b / (3 + c)* 2) – 1
  printf ("x = % f", x);
  printf ("y = % f", y);
  printf ("z = % f", z);
}
```

Output :

```
x = 10.00
y = 7.00
z = 4.00
```


Standards and formatted IOS:

Formatted Input

The function scanf() is used for formatted input from standard input and provides many of the conversion facilities of the function printf().

Syntax :

scanf (format, num1, num2,.....);

The function scnaf() reads and converts characters from the standards input depending on the format specification string and stores the input in memory locations represented by the other arguments (num1, num2,....).

For Example :

scanf(%c %d",&Name, &RollNo);

- **Note :** the data names are listed as &Name and &RollNo instead of Name and Roll No respectively. This is how data names are specified in a scanf() function. In case of string type data names, the data name is not preceded by the character &.

Example with program

Write a function to accept and display the element number and the weight of a proton. The element number is an integer and weight is fractional.

Solve here :

```
#include<stdio.h>
#include<conio.h>
main()
{
int e_num;
float e_wt;
printf ("Enter the Element No. and Weight of a Proton\n");
scanf ("%d %f"J&e_num, &e_wt);
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:2
Storage size for char data type:1
Storage size for float data type:4
Storage size for double data type:8
```

Expression

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Algebraic Expression	C Expression
a× b – c	a * b – c
(m + n)(x + y)	(m + n) * (x + y)
(ab/c)	a * b / c
$3x^2 + 2x + 1$	3*x*x+2*x+1
(x/y) + c	x/y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Example of evaluation statements are

```
x = a * b – c
y = b / c * a
z = a – b / c + d;
```

The following program illustrates the effect of presence of parenthesis in expressions.

```
main ()
{
  float a, b, c x, y, z;
  a = 9;
     b = 12;
     c = 3;
     x = a – b / 3 + c * 2 – 1;
     y = a – b / (3 + c) * (2 – 1);
```

3 Control Structures

Conditional Statements in C programming are used to make decisions based on the conditions. Conditional statements execute sequentially when there is no condition around the statements. If you put some condition for a block of statements, the execution flow may change based on the result evaluated by the condition. This process is called decision making in 'C.'

"The Control Statements are used for controlling the Execution of the program"

There are four types of control statements in C:

1. Decision making statements
2. Selection statements
3. Iteration statements
4. Jump statements

Derision Making Statement

If-se Statement

It is one of the powerful conditional statement. If statement is responsible for modifying the flow of execution of a program. If statement is always used with a condition. The condition is evaluated first before executing any statement inside the body of If. The syntax for if statement is as follows:

```
if (condition)
  instruction;
```

The condition evaluates to either true or false. True is always a non-zero value, and false is a value that contains zero. Instructions can be a single instruction or a code block enclosed by curly braces {}.

Example :

```
include<stdio.h>
int main()
{
      int num1 = 1;
      int num2 = 2;
      if(num1< num2)       //test-condition
      {
          printf("numl is smaller than num2");
      }
      return 0;
}
```

Output :

num1 is smaller than num2

The above program illustrates the use of if construct to check equality of two numbers.

```
#include<stdio.h>
int main()
{
    int num1=1;        ❶
    int num2=2;
    if (num1<num2)     ❷
    {
    ❸ printer("num1 is smaller than num2");
    }
    return 0;
}
```

1. In the above program, we have initialized two variables with numl, num2 with value as 1, 2 respectively.
2. Then, we have used if with a test-expression to check which number is the smallest and which number is the largest. We Kane used a relational expression in if construct. Since the value of numl is smaller than num2, the condition will evaluate to true.
3. Thus it will print the statement inside the block of If. After that, the control will go outside of the block and program will be terminated with a successful result.

If-else statement

The if-else statement is used to carry out a logical test and then take one of two possible actions depending on the outcome of the test (ie, whether the outcome is true or false).

Syntax :

```
if (condition)
{
  statements
}
  else
{
  statements
}
```

If the condition specified in the if statement evaluates to true, the statements inside the if-block are executed and then the control gets transferred to the statement immediately afterthe if-block. Even if the condition is false and no else-block is present, control gets transferred to the statement immediately after the if-block.

The else part is required only if a certain sequence of instructions needs to be executed if the condition evaluates to false. It is important to note that the condition is always specified in parentheses and that it is a good practice to enclose the statements in the if block or in the else-block in braces, whether it is a single statement or a compound statement.

The following program checks whether the entered number is positive or negative.

```
#include<stdio.h>
int main( )
{
int a;
printf(''inter value of a'');
scanf(''%d'',&a);
if(a<0)
}
   printf( ''\n The number %d is positive.'',a);
}
else
{
   printf(''\n The number %d is negative.'',a);
}
return 0;
}
```

The following program compares two strings to check whether they are equal or not.

```
#include<stdio.h>
#include <string.h>
int main()
{
  char a[20], b[20];
  printf(''\n Enter the first string:'');
  scanf(''%s'', a);
  printf(''\n Enter the second string:'');
  scanf(''%s'', b);
  if((strcmp(a, b)==0))
  {
     printf(''\nStrings are the same'');
  }
else
  {
  printf(''\nStrings are different'');
  }
return 0;
}
```

The above program compares two strings to check whether they are the same or not. The strcmp function is used for this purpose. It is declared in the string.h file as:

int strcmp(const char *sl, const char *s2);

It compares the string pointed to by si to the string pointed to by s2. The strcmp function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by si is greater than, equal to, or less than the string pointed to by s2.

Therefore in the above program, if the two strings a and b are equal, the strcmp function should return a 0. If it returns a 0, the strings are the same; else they are different.

Nested if and if-else Statements

It is also possible to **embed** or to **nest** if-else statements one within the other. Nesting is useful in situations where one of several different courses of action need to be selected.

The general format of a nested if-else statement is :

```
if (condition1)
{
//statement(s);
}
else if(condition2)
{
//statement(s);
}
.
.
.
.
else if (conditionN)
{
//statement(s);
}
else
{
//statement(s);
}
```

The above is also called the **if-else ladder**. During the execution of a nested if-else statement, as soon as a condition is encountered which evaluates to true, the statements associated with that particular if-block will be executed and the remainder of the nested if-else statements will be bypassed. If neither of the conditions are true, either the last else-block is executed or if the else-block is absent, the control gets transferred to the next instruction present immediately after the else-if ladder.

The following program makes use of the nested if-else statement to find the greatest of three numbers.

```
#include<stdio.h>
int main()
{
    int a, b,c;
    a = 6, b= 5, c=10;
    if (a > b)
        {
            if (b > c)
                {
```



```
                printf("\nGreatest is:", a);
                }
            else if(c > a)
                {
                printf("\nGreatest is:", c);
                }
        }
    else if(b>c) //outermost if-else block
        {
            printf("\nGreatest is:", b);
        }
    else
        {
            printf( "\nGreatest is:", c);
        }
    return 0;
}
```

The above program compares three integer quantities, and prints the greatest. The first if statement compares the values of a and b. If a>b is true, program control gets transferred to the if-else statement nested inside the if block, where b is compared to c. If b>c is also true, the value of a is printed; else the value of c and a are compared and if c>a is true, the value of c is printed. After this the rest of the if-else ladder is bypassed.

However, if the first condition a>b is false, the control directly gets transferred to the outermost else-if block, where the value of b is compared with c (as a is not the greatest).

If b>c is true the value of b is printed else the value of c is printed. Note the nesting, the use of braces, and the indentation. All this is required to maintain clarity.

Selection Statement

Switch-case Statement : A switch statement is used for **multiple way selections** that will branch into different code segments based on the value of a variable or expression. This expression or variable must be of integer data type.

Syntax :

```
switch (expression)
{
  case value1:
     code segment1;
     break;
  case value2:
     code segment2;
     break;
  .
  .
  .
```

```
    case valueN:
        code segmentN;
    break;
  default:
    default code segment;
  }
```

The value of this **expression** is either generated during program execution or read in as user input. The case whose value is the same as that of the **expression** is selected and executed. The optional default label is used to specify the code segment to be executed when the value of the expression does not match with any of the case values.

The break statement is present at the end of every case. If it were not so, the execution would continue on into the code segment of the next case without even checking the case value. For example, supposing a switch statement has five cases and the value of the third case matches the value of expression. If no break statement were present at the end of the third case, all the cases after case 3 would also get executed along with case 3. If break is present only the required case is selected and executed; after which the control gets transferred to the next statement immediately after the switch statement. There is no break after default because after the default case the control will either way get transferred to the next statement immediately after switch.

Example : a program to print the day of the week.

```
#include<stdio.h>
int main()
{
int day;
printf("\nEnter the number of the day:");
scanf("%d",&day);
switch(day)
  {
    case 1 :
       printf("Sunday");
       break;
    case 2 :
       printf("Monday");
       break;
    case 3 :
       printf("Tuesday");
       break;
    case 4 :
       printf("Wednesday");
       break;
    case 5 :
       printff "Thursday");
       break;
    case 6 :
```



```
       printf("Friday");
       break;
    case 7 :
       printf("Saturday");
       break;
    default:
       printf("Invalid choice");
  }
return 0;
  }
```

This is a very basic program that explains the working of the switch-case construct. Depending upon the number entered by the user, the appropriate case is selected and executed. For example, if the user input is 5, then case 5 will be executed. The break statement present in case 5 will pause execution of the switch statement after case 5 and the control will get transferred to the next statement after switch, which is:

```
return 0;
```

It is also possible to embed compound statements inside the case of a switch statement. These compound statements may contain control structures. Thus, it is also possible to have a nested switch by embedding it inside a case.

All programs written using the switch-case statement can also be written using the if-else statement. However, it makes good programming sense to use the if statement when you need to take some action after evaluating some simple or complex condition which may involve a combination of relational and logical expressions (*eg*, (if ((a! = 0)&&(b>5))).

If you need to select among a large group of values, a switch statement will run much faster than a set of nested ifs. The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression.

The switch statement must be used when one needs to make a choice from a given set of choices. The switch case statement is generally used in **menu-based applications**. The most common use of a switch-case statement is in data handling or file processing. Most of file handling involves the common functions: creating a file, adding records, deleting records, updating records, printing the entire file or some selective records. The following program gives an idea of how the switch case statement may be used in data handling.

Example : A switch case statement used in data file processing.

```
#include<stdio.h>
int main()
{ //create file &set file pointer .
int choice;
printf("\n Please select from the following options:");
printf("\n 1. Add a record at the end of the file.");
printf("\n 2. Add a record at the beginning of the file:");
printf("\n 3. Add a record after a particular record:";
printf("\nPlease enter your choice:(1/2/3)?");
scanf("%d",&choice);
switch(choice)
```

```
    {
      case 1 :
          //code to add record at the end of the file
          break;
      case 2 :
          //code to add record at the beginning of the file
          break;
      case 3 :
          //code to add record after a particular record
          break;
      default:
          printf("\n Wrong Choice");
    }
  return 0;
  }
```

The above example of switch-case generally involves nesting the switch-case construct inside an iteration construct like do-while.

Loop

A Computer is used for performing many Repetitive types of tasks The Process of Repeatedly performing tasks is known as looping .The Statements in the block may be Executed any number of times from Zero to Up to the Condition is True. The Loop .is that in which a task is repeated until the condition is true or we can say in the loop will Executes all the statements are until the given condition is not to be false.

These are generally used for repeating the statements., In this There is Either Entry controlled loop or as Exit Controlled Loop. We know that before Execution of State Conditions are Checked these are Performed by Entry Controlled Loops Which First Checks condition And in Exit Controlled Loop it Checks Condition for Ending Loop. Whether given Condition is False or not if a Loop First Checks Condition For Execution then it is called as Entry Controlled Loop and if a Loop Checks Condition after the Execution of Statement then they are called as Exit Controlled Loops.

In the loop generally there are three basic operations are performed :

1. Initialization
2. Condition check
3. Increment / decrement(update)

Types of loop

1. while loop
2. do-while loop
3. for loop

1. while

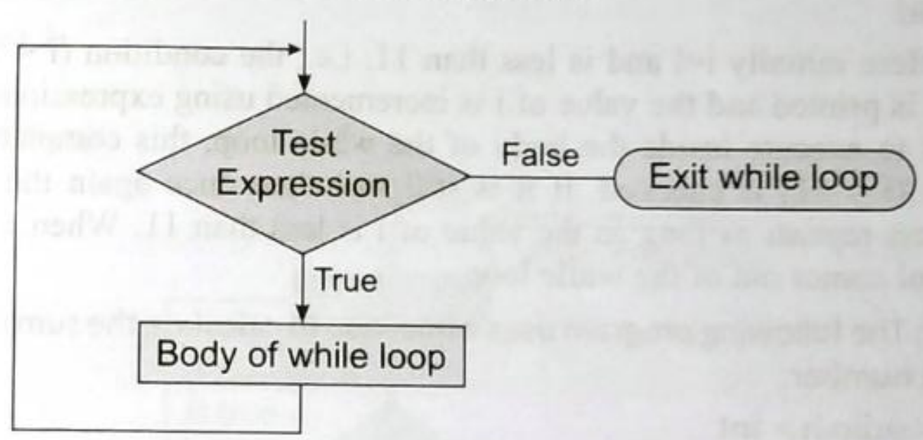
while Loop is Known as Entry Controlled Loop because in The while loop first we initialize the value of variable or Starting point of Execution and then we check the condition and if the condition is true then it will execute the statements and then after it increments or decrements the value of a variable. But in the while loop if a Condition is false then it will never. Executes toe Statement So that For Execution, this is must that the Condition must be true.

The syntax of C while loop is as follows :

```
while (expression)
{
// execute statements
}
```

The while loop executes as long as the given logical expression evaluates to true. When expression evaluates to false, the loop stops. The expression is checked at the beginning of each iteration. The execute statements inside the body of the while loop statement are not executed if the expression evaluates to false when entering the loop. It is necessary to update the loop condition inside the loop body to avoid an indefinite loop.

The following flowchart illustrates the while loop in C :



Steps of while loops are as follows :

1. The test condition is evaluated and if it is true, the statement block is executed.
2. On execution of the body, test condition is repetitively checked and if it is true the statement block is executed.
3. The process of execution of the statement block will be continued until the test condition becomes false. Therefore you must always include a statement which alters the value of the condition so that it ultimately becomes false at some, point. Note that if the condition is never updated and the condition never becomes false, then the computer will run into an infinite loop which is never desirable.
4. The control is transferred out of the loop.

- **Example 1 :** The following program uses while loop to print "Hello World" 10 times.

```
#include<stdio.h>
int main(void)
{
   int i = 1;
      /* while loop execution */
   while(i < 11)
   {
      printf("Hello World \n");
      i++; /* statement that change the value of condition */
   }
   return 0;
```

```
}
```

Output :

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

Explanation : Here initially i=l and is less than 11, i.e., the condition (i < 11) is true, so in the while loop ' World" is printed and the value of i is incremented using expression i++. As there are no more statements left to execute inside the body of the while loop, this completes the first iteration. Again the condition (i < 11) is checked, if it is still true then once again the body of the loop is executed. This process repeats as long as the value of i is less than 11. When i reaches 11, the loop terminates and control comes out of the while loop.

- **Example 2 :** The following program uses while loop to calculate the sum of individual digits of an entered number.

```
#include <stdio.h> int
main(void)
{
  int n, num, sum = 0, remainder;
  printf("Enter a number: ");
  scanf("%d", &n);
  num = n;
  while( num >0)
  {
     remainder = num % 10; /* take out the last digit of a number */
     sum = sum + remainder; /* add the remainder to the sum */
     num = num / 10;
  }
  printf("\nSum of digits of %d = %d", n, sum);
  return 0;
}
```

Output :

```
Enter a number: 437
Sum of digits of 437 = 14
```

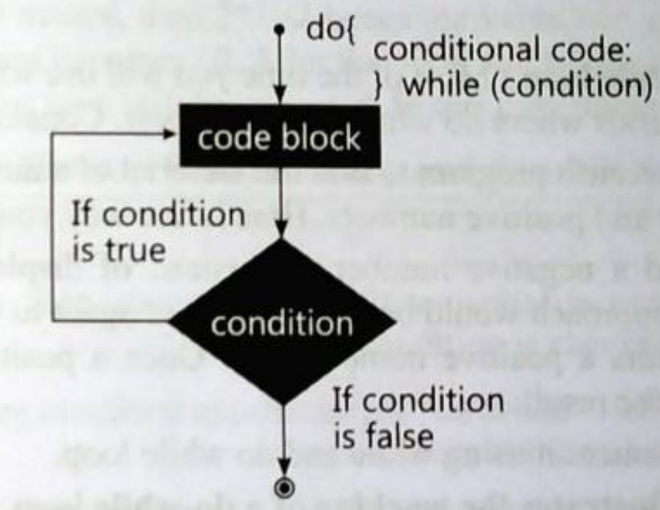
Explanation : Here we are extracting the digits of the •••her from right to left and then these digits are added one by one to the variable sum. Mote that the variable sum is initialized to 0. This is because we are adding some numbers to it. and if not initialized then these numbers will be added to garbage value present in it.

2. do-while

This is Also Called as Exit Controlled Loop we know that in The while loop the condition is check before the execution of the program but if the condition is not true then it will not execute the statements so for this purpose we use the do while loop in this first it executes the statements and then it increments the value of a variable anc the condition So in this either the condition is true or not it Execute the stat time.

Syntax :

```
do
{
  // body of do while loop
  statement 1;
  statement 2;
}
while(condition);
```



In do while loop first the statements in the body are executed then the condition is checked. If the condition is true then once again statements in the body are executed. This process keeps repeating until the condition becomes false. As usual, if the body of cfb while loop contains only one statement then braces ({}) can be omitted. Notice that unlike the while loop, in do while a semicolon(;) is placed after the condition.

The do while loop differs significantly from the while loop because in do while loop statements in the body are executed at least once even if the condition is false. In the case of while loop the condition is checked first and if it true only then the statements in the body of the loop are executed.

```
#include <stdio.h>
int main ()
{
  /* local variable definition */
  int a = 10;
  /* do loop execution */
  do
{
  printf("value of a: %d\n"/ a);
  a = a + 1;
```

```
}
while( a < 20 );
return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Where should I use do while loop : Most of the time you will use while loop instead of do while. However, there are some scenarios where do while loop suits best. Consider the following problem.

Let's say the you want to create a program to find the factorial of a number. As you probably know that factorial is only valid for 0 and positive numbers. Here is one way you can approach this problem.

Let's say the user entered a negative number, so instead of displaying an error message and quitting the program, a better approach would be to ask the user again to enter a number. You have to keep asking until the user enters a positive number or 0. Once a positive number or 0 is entered, calculate factorial and display the result.

Let's see how we can implement it using while and do while loop.

The following program illustrates the working of a do-while loop :

We are going to print a table of number 2 using do while loop.

```
#include<stdio.h>
#include<conio.h>
int main()
{
  int num=l;      //initializing the variable
  do          //do-while loop
  {
            printf("%d\n";2*num);
            num++;      //incrementing operation
  }
while(num<=10);
return 0;
}
```

Output :

```
2
4
```



```
6
8
10
12
14
16
18
20
```

In the above example, we have printed multiplication table of 2 using a do-while loop. Let's see how the program was able to print the series,

1. First, we have initialized a variable 'num' with value 1. Then we have written a loop.
2. In a loop, we have a print function that will print the series by multiplying the value of num with 2.
3. After each increment, the value of num will increase by 1, and it will be printed on the screen.
4. Initially, the value of num is 1. In a body of a loop, the print function will be executed in this way: 2*num where num=l, then 2*1=2 hence the value two will be printed. This will go on until the value of num becomes 10. After that loop will be terminated and a statement which is immediately after the loop will be executed. In this case return 0.

Difference between while and do while loop

while	do-while
Condition is checked first then statement(s) is executed.	Statement(s) is executed atleast once, thereafter condition is checked.
In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Takes much less time to execute however and the code is shorter.	Takes extra time to execute and code turns into longer.
It might occur statement(s) is executed zero times, If condition is false.	At least once the statement(s) is executed.
No semicolon at the end of while.	Semicolon at the end of while.
while(condition)	while(condition);
If there is a single statement, brackets are not required.	Brackets are always required.
Variable in condition is initialized before the execution of loop.	Variable may be initialized before or within the loop.
while loop is entry controlled loop.	do-while loop is exit controlled loop.
while(condition) {statement(s);}	do {statement(s);} while (condition);

3. for

In This loop all the basic operations like initialization, condition checking and incrementing or

decrementing all these are performing its execution but only different in its syntax.

The syntax of the for loop is :

```
for (initialization Statement; test Expression; update Statement)
    {
      // statements inside the body of loop
    }
```

How for loop works :

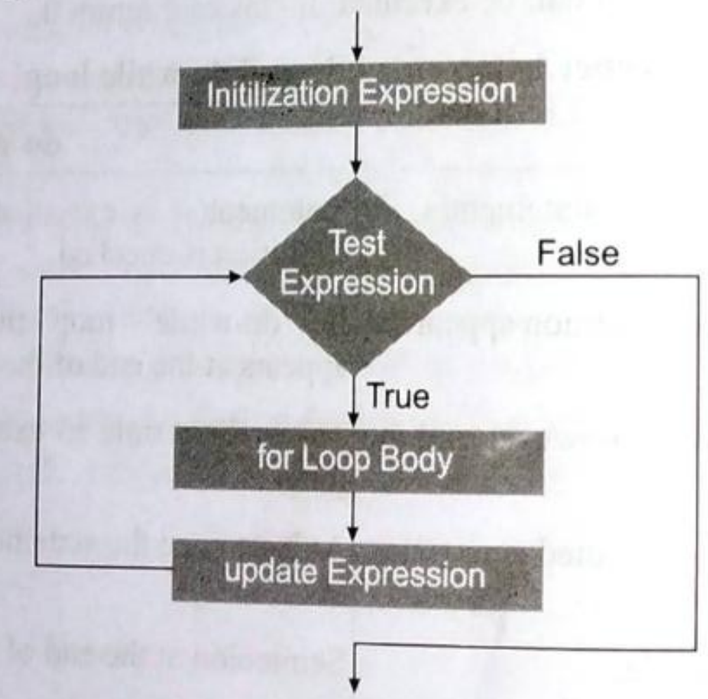
- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.

However, if the test expression is evaluated to true, statements inside the body of for loop are executed, and the update expression is updated.

Again the test expression is evaluated.

This process goes on until the test expression is false. When the test expression is false, the loop terminates.

To learn more about test expression (when the test expression is evaluated to true and false), check out relational and logical operators.



- **Example 1 : for loop**

```
// Print numbers from 1 to 10
#include <stdio.h>
int main()
{
  int i;
  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
}
```


Output :

1 2 3 4 5 6 7 8 9 10

1. i is initialized to 1.
2. The test expression i < 11 is evaluated. Since 1 less than 11 is true, the body of for loop is executed. This will print the 1 (value of i) on the screen.
3. The update statement ++i is executed. Now, the value of i will be 2. Again, the test expression is evaluated to true, and the body of for loop is executed. This will print 2 (value of i) on the screen.
4. Again, the update statement ++i is executed and the test expression i < 11 is evaluated. This process goes on until i becomes 11.
5. When i becomes 11, i < 11 will be false, and the for loop terminates.

- **Example 2 : for loop**

```
// Program to calculate the sum of first n natural numbers
// Positive integers 1,2,3...n are known as natural numbers
#include <stdio.h>
int main()
{
   int num, count, sum = 0;
   printf("Entera positive integer:");
   scanf("%d", &num);
   // for loop terminates when num is less than count
   for(count = 1; count <= num; ++count)
   {
      sum += count;
   }
   printf("Sum = %d", sum);
   return 0;
}
```

Output : Enter a positive integer :

10 Sum = 55

The value entered by the user is stored in the variable num. Suppose, the user entered 10. The count is initialized to 1 and the test expression is evaluated. Since the test expression count<=num (1 less than or equal to 10) is true, the body of for loop is executed and the value of sum will equal to 1.

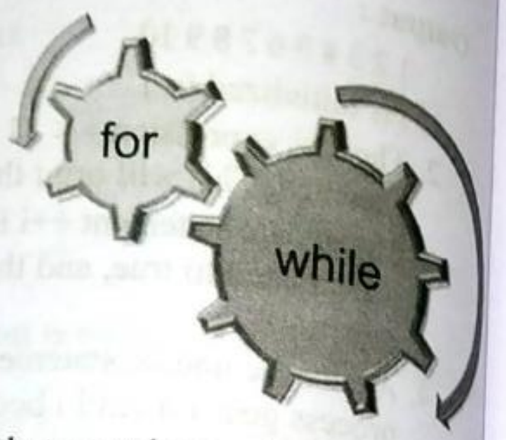
Then, the update statement ++count is executed and the count will equal to 2. Again, the test expression is evaluated. Since 2 is also less than 10, the test expression is evaluated to true and the body of for loop is executed. Now, the sum will equal 3.

This process goes on and the sum is calculated until the count reaches 11.

When the count is 11, the test expression is evaluated to 0 (false), and the loop terminates.

Then, the value of sum is printed on the screen.

Difference between for and while loop



In C+ + and Java, the iteration statements, for loop, while loop and do-while loop, allow the set of instructions to be repeatedly executed, till the condition is true and terminates as soon as the condition becomes false. Conditions in iteration statements may be predefined as in for loop or open-ended as in while loop.

There are several 'for' loop variations in C are implied to increase its applicability, power and flexibility. For example, the for loop allows us to use more than one variable inside the loop in order to control it, and the use of converge function with 'for' loop. Conversely, with while loop we can not use many variations, that must be used with the standard syntax.

There are some major differences between for and while loops, which are explained further with the help of a comparison chart.

Comparison Chart

Basis for comparison	for	while
Declaration	for (initialization; condition; iteration) { //body of 'for' loop }	while (condition) { statements; //body of loop }
Format	Initialization, condition checking, iteration statement is written at the top of the loop.	Only initialization and condition checking is done at the top of the loop.
Use	The 'for' loop used only when we already knew the number of iterations.	The 'while' loop used only when the number of iteration are not exactly known.
Condition	If the condition is not put up in 'for' loop, then loop iterates infinite times.	If the condition is not put up in 'while' loop, it provides compilation error.
Initialization	In 'for' loop the initialization once done is never repeated.	In while loop if initialization is done during condition checking, then initialization is done each time the loop iterate.
Iteration statement	In 'for' loop iteration statement is written at top, hence, executes only after all statements in loop are executed.	In 'while' loop, the iteration statement can be written anywhere in the loop.

goto statement :

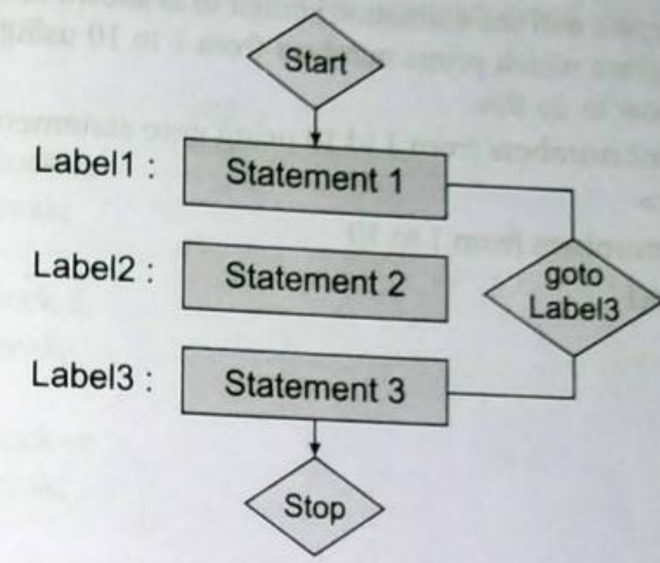
The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.

Syntax :

goto label;

..
.
label: statement;

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.



Below are some examples on how to use goto statement:

- **Examples :**

❖ **Type 1:** In this case, we will see a situation similar to as shown in Syntaxl above. Suppose we need to write a program where we need to check if a number is even or not and print accordingly using the goto statement. Below program explains how to do this:

```
// C program to check if a number is even or not using goto statement
#include<stdio.h>
  // function to check even or not
void checkEvenOrNot (int num)
  if (num % 2 == 0)
     //jump to even
     goto even;
  else
     //jump to odd
     goto odd;
  even:
     printf("%d is even", num);
     // return if even
     return;
  odd:
     printf("%d is odd", num);
}
int main()
{
  int num = 26;
```

```
    checkEvenOrNot(num);
    return 0;
}
```

Output :

26 is even

❖ **Type 2 :** In this case, we will see a situation similar to as shown in Syntax 1 above. Suppose we need to write a program which prints numbers from 1 to 10 using the goto statement. Below program explains how to do this.

```
// C program to print numbers from 1 id 10 using goto statement
#include <stdio.h>
//function to print numbers from 1 to 10
void printNumbers()
{
  int n = 1;
label:
  printf("%d ",n);
  n++;
  if(n<=10)
    goto label;
}
// Driver program to test above function
int main()
{
  printNumbers();
  return 0;
}
```

Output :

1 2 3 4 5 6 7 8 9 10

Disadvantages of using goto statement

- ❖ The use of goto statement is highly discouraged as it makes the program logic very complex.
- ❖ use of goto makes the task of analyzing and verifying the correctness of programs (particularly those involving loops) very difficult.
- ❖ Use of goto can be simply avoided using break and continue statements.
- ❖ In modern programming, goto statement is considered a harmful construct and a bad programming practice.
- ❖ The goto statement can be replaced in most of C program with the use of break and continue statements.
- ❖ In fact, any program in C programming can be perfectly written without the use of goto statement.
- ❖ All programmer should try to avoid goto statement as possible as they can.

Switch

Switch statement in C tests the value of a variable and compares it with multiple cases. Once the

case match is found, a block of statements associated with that particular case is executed. Each case in a block of a switch has a different name/number which is referred to as an identifier. The value provided by the user is compared with all the cases inside the switch block until the match is found.

If a case match is NOT found, then the default statement is executed, and the control goes out of the switch block.

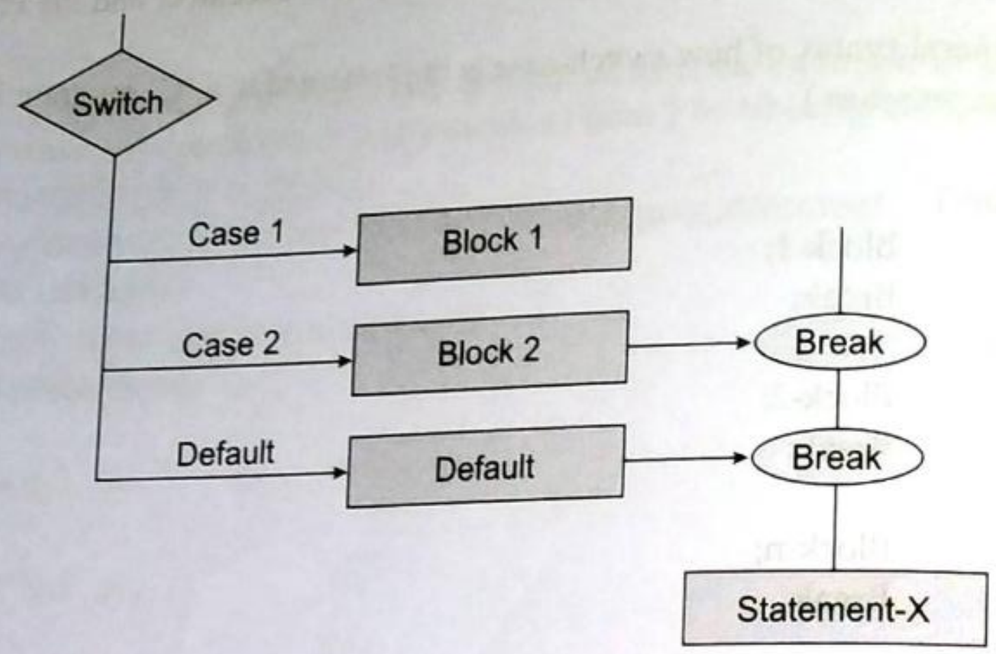
Syntax : A general syntax of how switch-case is implemented in a 'C' program is as follows :

```
switch( expression )
{
case value-1:
            Block-1;
            Break;
case value-2:
            Block-2;
            Break;
case value-n:
            Block-n;
            Break;
default:
            Block-1;
            Break;
}
Statement-x;
```

- ❖ The expression can be integer expression or a character expression.
- ❖ Value-1, 2, n are case labels which are used to identify each case individually. Remember that case labels should not be same as it may create a problem while executing a program. Suppose we have two cases with the same label as '!'. Then while executing the program, the case that appears first will be executed even though you want the program to execute a second case. This creates problems in the program and does not provide the desired output.
- ❖ case labels always end with a colon (:). Each of these cases is associated with a block.
- ❖ A block is nothing but multiple statements which are grouped for a particular case.
- ❖ Whenever the switch is executed, the value of test-expression is compared with all the cases which we have defined inside the switch. Suppose the test expression contains value 4. This value is compared with all the cases until case whose label four is found in the program. As soon as a case is found the block of statements associated with that particular case is executed and control goes out of the switch.
- ❖ The break keyword in each case indicates the end of a particular case. If we do not put the break in each case then even though the specific case is executed, the switch in C will continue to execute all the cases until the end is reached. This should not happen; hence we always have to put break keyword in each case. Break will terminate the case once it is executed and the control will fall out of the switch.
- ❖ The default case is an optional one. Whenever the value of test-expression is not matched with any of the cases inside the switch, then the default will be executed. Otherwise, it is not necessary to write default in the switch.
- ❖ Once the switch is executed the control will go to the statement-x, and the execution of a program will continue.

Flow Chart of switch Case

Following diagram illustrates how a case is selected in switch case:



- **Example :** Following program illustrates the use of switch :

```
#include<stdio.h>
  int main()
  {
     int num = 8;
     switch (num)
     {
     case 7:
            printf("Value is 7");
            break;
       case 8:
            printf("Value is 8");
            break;
      case 9:
            printf("Value is 9");
            break;
      default:
            printf("Out of range");
            break;
   }
  return 0;
}
```

Output :

Value is 8

1. In the given program we have explain initialized a variable num with value 8.
2. A switch construct is used to compare the value stored in variable num and execute the block of statements associated with the matched case.
3. In this program, since the value stored in variable num is eight, a switch will execute the case whose case-label is 8. After executing the case, the control will fall out of the switch and program will be terminated with the successful result by printing the value on the output screen.

Try changing the value of variable num and notice the change in the output. For example, we consider the following program which defaults:

```
#include <stdio.h>
int main()
{
int language = 10;
   switch (language)
   {
   case 1:
      printf("C#\n");
      break;
   case 2:
      printf("C\n");
      break;
  case 3:
      printf("C++\n");
      break;
  default:
      printf("0ther programming language\n");
   }
}
```

Output :

Other programming language

Break statement

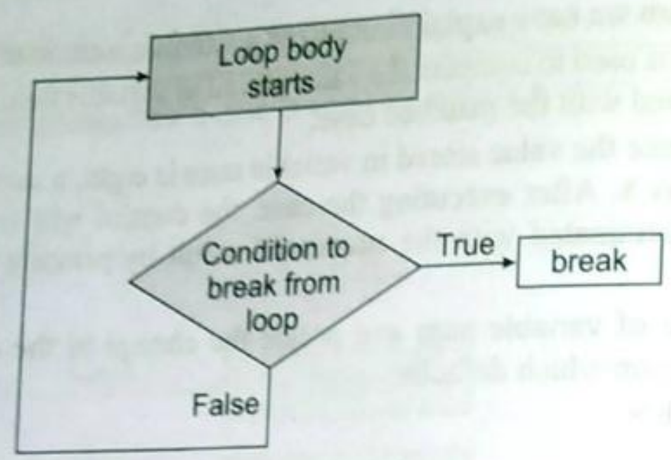
The break statement is used inside loop or switch statement. When compiler finds the break statement inside a loop, compiler will abort the loop and continue to execute statements followed by loop.

In general the break statements we used in the situations where we need to stop the loop execution based on the condition or not sure how many times the loop is to be iterate. If the break statements using inside the nested loop, then the break statement breaks the inner loop and starts executing the statement after the inner loop of the program control continue to the outer loop.

Syntax :

break;

Basically break statements are used in the situations when we are not sure about the actual number of iterations for the loop or we want to terminate the loop based on some condition.



There are two usages and the given statement is explained below :

- **Inside a Loop :** If the break statement is using inside a loop along with the if statement then if the condition becomes true the loop is immediately terminated and the next statement after the loop starts executing by the program control.
- Reak statement inside while loop
- Consider the following example to use the break statement inside while loop.

- **Example :**

```
tfinclude <stdio.h>
int main ()
{
int co = 0;
while(co < 10)
{
printf("loop%d\n",co);
if(co == 6)
break;
else
co = co +1;
}
printf("\n",co);
printf("The loop terminal at co = %d", co);
return 0;
}
```

The output of the above code :

```
loop 0
loop 1
loop 2
loop 3
loop 4
loop 5
loop 6
Hie loop terminat at co = 6
```


- **Example : Break statement inside the do-while loop:**

Consider the following example to use the break statement with a do-while loop.

```
#include<stdio.h>
int main ()
{
int co = 0;
do
{
printf('loop %d\n",co);
if(co == 6)
break;
else
co = co+1;
}
while(co < 10);
printf("\n",co);
printf("The loop terminate at co = %d", co);
return 0;
}
```

The output of the above code :

```
loop 0
loop 1
loop 2
loop 3
loop 4
loop 5
loop 6
Che loop terminat at co = 6
```

Inside a Switch Case

If Break Statement in C is using inside a switch case after each switch case then the break statement terminates a case after executing the case.

- **Example :** Break statement inside the switch case

```
#include <stdio.h>
void main()
{
char opt;
printf("Enter the option \"A\", \"B\", \"C\" or \"D\" :");
scanf( "%c", &opt );
switch (opt)
{
case 'B':
```

```
printf( "You have entered option B " );
break;
case 'A':
printf( "You have entered option A " );
break;
case 'D':
printf( "You have entered option D ");
break;
case 'C':
printf( "You have entered option C " );
break;
default:
printf ( "You have not entered option A, B, C, or D, wrong input ");
}
}
```

The output of the above code :

Enter the option "A", "B", "C" or "D" : A You have entered option A
Enter the option "A", "B", "C" or "D" : H
You have not entered option A, B, C, or D, wrong input

- Example :

```
#include<stdio.h>
int main ()
{
/* local variable definition */
int a = 10;
/* while loop execution */
while( a < 20 )
{
      printf("value of a: %d\n", a);
      a++;
 if( a > 15)
 {
    /* terminate the loop using break statement */
    break;
 }
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result :

value of a: 10
value of a: 11
value of a: 12

value of a: 13
value of a: 14
value of a: 15

continue

This statement is majorly used in the case of iterators or in case of looping. This statement as the name already suggests, makes sure that the code continues running after a particular statement is executed. It is used in the same way as the break statement, but the break statement would stop the execution of the loop or series of statements, but the continue statement in return would continue the execution of the code.

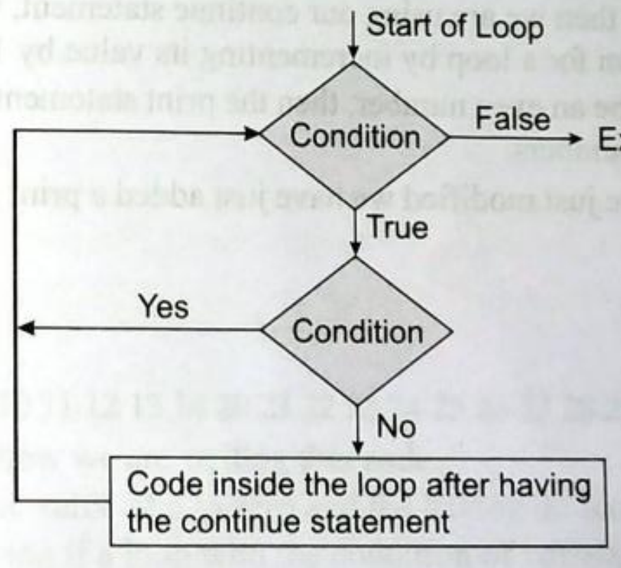
Below is the syntax for the continue statement,

Syntax :

continue;

As already mentioned, the continue statement is used in loops. So the only syntax for the statement would be like above.

Flow Chart : We can understand it better through a flow chart, let's see it below.



Explanation :

- As already known, any loop starts with a condition, and there would be two scenarios for it. One is the statement that has to be executed when a condition is true and others when it is false.
- When a condition is false, it is going to obviously exit the loop.
- And when a condition is true, and have our continue statement, the iterator again goes back to the condition and the above process continues.
- If the condition does not have that continue statement, then the code below is executed.

- **Example :** Finding odd numbers from o to 20.

Code :

```
#include <stdio.h>
int main()
{
int i;
for(i=0;i<20;i++)
```

```
{
if(i%2==0)
{
continue;
}
printf("%d ",i);
}
}
```

Output :

1 3 5 7 9 11 13 15 17 19

As per the observation, we can see how the program works:

- We declared a variable i.
- We made for a loop by initializing the value of I to 0 and incrementing it by one till the number is less than 20.
- And then we have another condition that if, modulo division of I with 2 is zero; that is it would denote an even number, then we are using our continue statement, which is, in turn, iterating the program back to them for a loop by incrementing its value by 1.
- If the variable i will not be an even number, then the print statement is being executed, which in turn prints only odd numbers.

For the above program, we have just modified we have just added a print statement below continue statement.

Code :

```
#include<stdio.h>
int main()
{
int i;
for(i=0;i<20;i++)
{
if(i%2==0)
{
continue;
printf("This will not be executed");
}
printf("%d ",i);
}
}
```

Output :

1 3 5 7 9 11 13 15 17 19

The same output as the first example program is obtained. Through this change, we can tell that after the continue statement is encountered; the iteration directly goes above again. Any statement to the immediate below or continue statement present in the same loop or if/else condition will not be executed.

- **Example :** Let a movie theater has 30 seats and 5 seats from 15th seat are booked, so how can we show the remaining seats to people.

We are trying to write this using a do-while loop and we can write in a similar way as above just to display the numbers.

Code :

```
#include <stdio.h>
int main ()
{
int a = 0;
/* do loop execution */
do
{
if( a == 15)
{
a = a + 5;
continue;
}
printf("%d ", a);
a++;
} while(a<30);
return 0;
}
```

Output :

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 20 21 22 23 24 25 26 27 28 29

These are the steps on how we are writing this code.

- We initialized the value of a to zero and the having do loop.
- Then we are having if a loop with the condition of variable a being equal to 15.
- Then incrementing the value of a by 5 and then using continue to start the loop again.
- Then we can get the numbers after 20 and then our while loop will check the value for 'a' value till 30 numbers.

Differentiate between break and continue statement

The major difference between break and continue statements in C language is that a break causes the innermost enclosing loop or switch to be exited immediately. Whereas, the continue statement causes the next iteration of the enclosing for, while, or do loop to begin. The continue statement in while and do loops takes the control to the loop's test-condition immediately, whereas in the for loop it takes the control to the increment step of the loop.

The continue statement applies only to loops, not to switch. A continue inside a switch inside a loop causes the next loop iteration.

Practically, break is used in switch, when we want to exit after a particular case is executed; and in loops, when it becomes desirable to leave the loop as soon as a certain condition occurs (for instance, you detect an error condition, or you reach the end of your data prematurely).

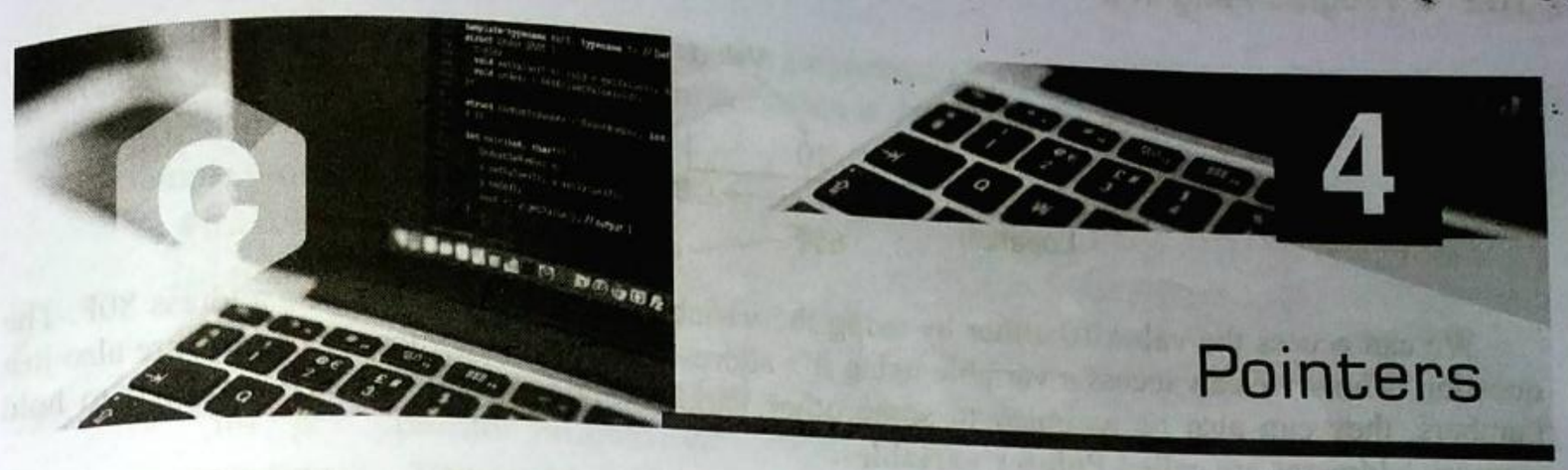
The continue statement is used when we want to skip one or more statements in loop's body and to

transfer the control to the next iteration.

Difference Between break and continue

Difference Between break and continue

Break	Continue
A break can appear in both switch and loop (for, while, do) statements.	A continue can appear only in loop (for, while, do) statements.
A break causes the switch or loop statements to terminate the moment it is executed. Loop or switch ends abruptly when break encountered.	A continue doesn't terminate the loop, it causes the loop to go to the next is iteration. All iterations of the loop are executed even if continue is encountered. The continue statement is used to skip statements is the loop that appear after the continue.
The break statement can be used both switch and loop statements.	The continue statement can appear only in loops. You will get an error if this appears in switch statement.
When a break statement is encountered, it terminates the block and gets the control out of the switch or loop.	When a continue statement is encountered, it gets the control to the next iteration of the loop.
A break causes the innermost enclosing loop or switch to be exited immediately.	A continue inside a loop nested within a switch causes the next loop iteration.
• **Example :** #include <stdio.h> int main() { int i; for (i=0; i<5; ++i) { if (i==3) break; printf("%d", i); } return 0; } **Output :** 0 1 2	• **Example :** #include <stdio.h> int main() { int i; for (i=0; i<5; ++i) { if (i==3) continue; printf("%d", i); } return 0; } **Output :** 0 1 2 4



Introduction to Pointers

A Pointer in C language is a variable which holds the address of another variable of same data type.

Pointers are used to access memory and manipulate the address.

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language.

Address in C

Whenever a variable is defined in C language, a memory location is assigned for it, in which it's value will be stored. We can easily check this memory address, using the & symbol.

If var is the name of the variable, then &var will give it's address.

Let's write a small program to see memory address of any variable that we define in our program.

```
#include<stdio.h>
void main()
{
   int var = 7;
   printf("Value of the variable var is: %d\n", var);
   printf("Memory address of the variable var is: %x\n", &var);
}
```

Value of the variable var is: 7

Memory address of the variable var is: bcc7a00

You must have also seen in the function scanf(), we mention &var to take user input for any variable var.

```
scanf("%d", &var);
```

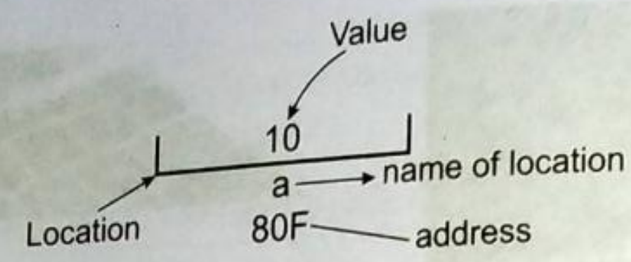
This is used to store the user inputted value to the address of the variable var.

Concept of Pointers

Whenever a variable is declared in a program, system allocates a location *i.e* an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

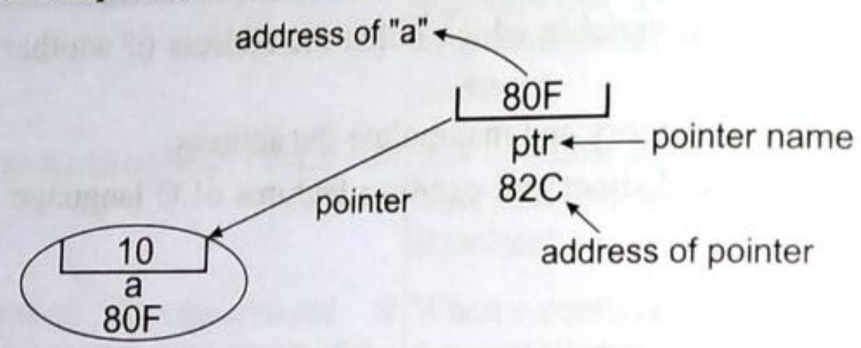
Let us assume that system has allocated memory location 80F for a variable a.

```
int a = 10:
```



We can access the value 10 either by using the variable name a or by using its address 80F. The question is how we can access a variable using it's address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables.**

A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** memory location.



Declaring a Pointer

Like variables, pointers have to be declared before they can be used in your program. Pointers can be named anything you want as long as they obey C's naming rules. A pointer declaration has the following form.

```
data_type * pointer_variable_name;
```

Here,

- **data_type** is the pointer's base type of C's variable types and indicates the type of the variable that the pointer points to.
- The asterisk (*: the same asterisk used for multiplication) which is indirection operator, declares a pointer.

Let's see some valid pointer declarations

```
int *ptr_thing;          /* pointer to an integer */
int *ptr1,thing;         /* ptr1 is a pointer to type integer and thing is an
                            integer variable */
double *ptr2;            /* pointer to a double */
float *ptr3;             /* pointer to a float */
char *ch1 ;              /* pointer to a character */
float *ptr, variable;    /*ptr is a pointer to type float and variable is an
                            ordinary float variable */
```

Initialize a Pointer

After declaring a pointer, we initialize it like standard variables with a variable address. If pointers are not uninitialized and used in the program, the results are unpredictable and potentially disastrous.

To get the address of a variable, we use the ampersand (&)operator, placed before the name of a variable whose address we need. Pointer initialization is done with the following syntax.

```
Pointer = &variable;
```

A simple program for pointer illustration is given below:

```
#include <stdio.h>
int main()
{
  int a=10;   //variable declaration
  int *p;     //pointer variable declaration
  p=&a;       //store address of variable a in pointer p
  printf("Address stored in a variable p is:%x\n",p); //accessing the address
  printf("Value stored in a variable p is:%d\n",*p); //accessing the value
  return 0;
}
```

Output :

```
Address stored in a variable p is :60ff08
Value stored in a variable p is:10
```

Operator	Meaning
*	Serves 2 purpose 1. Declaration of a pointer 2. Returns the value of the referenced variable
&	Serves only 1 purpose ❖ Returns the address of a variable

Advantages of using pointers

(i) Pointers make the programs simple and reduce their length.
(ii) Pointers are helpful in allocation and de-allocation of memory during the execution of the program. Thus, pointers are the instruments of dynamic memory management,
(iii) Pointers enhance the execution speed of a program,
(iv) Pointers are helpful in traversing through arrays and character strings. The strings are also arrays of characters terminated by the null character ('\O').
(v) Pointers also act as references to different types of objects such as variables, arrays, functions,structures, etc. However, C language does not have the concept of references as in C++. Therefore, in C we use pointer as a reference.
(vi) Pointers may be used to pass on arrays, strings, functions, and variables as arguments of a function.
(vii) Passing on arrays by pointers saves lot of memory because we are passing on only the address of array instead of all the elements of an array, which would mean passing on copies of all the elements and thus taking lot of memory space.
(viii) Pointers are used to construct different data structures such as linked lists, queues, stacks, etc.
(ix) Pointers are more efficient in handling Arrays and Structures.
(x) Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
(xi) It reduces length of the program and its execution time as well.

(xii) Pointers provide direct access to memory.

Drawbacks of Pointers in C

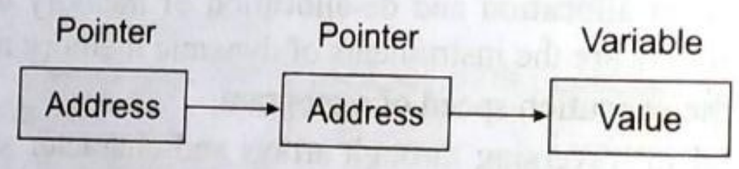
- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Key Points To Remember About Pointers In C

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int -

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example -

```
#include <stdio.h>
int main ()
{
  int var;
  int *ptr;
  int **pptr;
  var = 3000;
```



```
  /* take the address of var */
  ptr = &var;
  /* take the address of ptr using address of operator & */
  pptr = &ptr;
  /* take the value using pptr */
  printf("Value of var = %d\n", var);
  printf("Value available at *ptr = %d\n", *ptr);
  printf("Value available at **pptr = %d\n", **pptr);
  return 0;
}
```

When the above code is compiled and executed, it produces the following result :

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **ppstr = 3000
```

Types of a Pointer

Null Pointer

We can create a null pointer by assigning null value during the pointer declaration. This method is useful when you do not have any address assigned to the pointer. A null pointer always contains value 0.

Following program illustrates the use of a null pointer:

```
#include <stdio.h>
int main()
{
  int *p = NULL; //null pointer
  printf("The value inside variable p is:\n%x",p);
  return 0;
}
```

Output:

```
The value inside variable p is: 0
```

Void Pointer

In C programming, a void pointer is also called as a generic pointer. It does not have any standard data type. A void pointer is created by using the keyword void. It can be used to store an address of any variable.

Following program illustrates the use of a void pointer:

```
#include <stdio.h>
int main()
{
  void *p = NULL;     //void pointer
  printf("The size of pointer is:%d\n",sizeof(p));
  return 0;
```

```
}
Output:
The size of pointer is: 4
```

Wild Pointer

A pointer is said to be a wild pointer if it is not being initialized to anything. These types of pointers are not efficient because they may point to some unknown memory location which may cause problems in our program and it may lead to crashing of the program. One should always be careful while working with wild pointers.

Following program illustrates the use of wild pointer:

```
#include <stdio.h>
int main()
{
  int *p; //wild pointer
  printf("\n%d",*p);
  return 0;
}
```

Output

```
timeout: the monitored command dumped core
sh: line 1: 95298 Segmentation fault      timeout 10s main
```

Pointers and Arrays

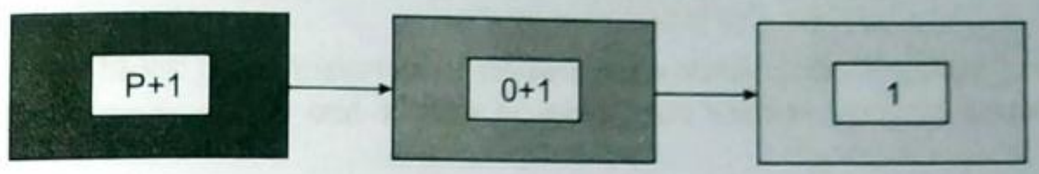
Traditionally, we access the array elements using its index, but this method can be eliminated by using pointers. Pointers make it easy to access each array element.

```
#include <stdio.h>
int main()
{
  int a[5]={1,2,3,4,5};  //array initialization
  int *p;    //pointer declaration
        /*the ptr points to the first element of the array*/

  p=a; /*We can also type simply ptr==&a[0] */

  printf("Printing the array elements using pointer\n");
  for(int i=0;i<5;i++)   //loop for traversing array elements
  {
      printf("\n%x",*p); //printing array elements
      p++;   //incrementing to the next element, you can also write p=p+1
  }
  return 0;
}
```

Output


```
1
2
3
4
5
```

Adding a particular number to a pointer will move the pointer location to the value obtained by an addition operation. Suppose p is a pointer that currently points to the memory location 0 if we perform following addition operation, p+1 then it will execute in this manner:



Since p currently points to the location 0 after adding 1, the value will become 1, and hence the pointer will point to the memory location 1.

Pointers and Strings

A string is an array of char objects, ending with a null character '\ 0'. We can manipulate strings using pointers. Here is an example that explains this section

```
#include <stdio.h>
#include <string.h>
int main()
{
  char str[]="Hello Guru99!";
  char *p;
  p=str;
  printf("First character is:%c\n",*p);
  p =p+1;
  printf("Next character is:%c\n",*p);
  printf("Printing all the characters in a string\n");
  p=str;  //reset the pointer
  for(int i=0;i<strlen(str);i++)
  {
     printf("%c\n",*p);
     p++;
  }
  return 0;
}
```

Output

```
First character is:H
Next character is:e
Printing all the characters in a string
H
```

e
l
l
o
G
u
r
u
9
9
!

Functions Pointers

Pointers give greatly possibilities to 'C' functions which we are limited to return one value. With pointer parameters, our functions now can process actual data rather than a copy of data.

In order to modify the actual values of variables, the calling statement passes addresses to pointer parameters in a function.

Functions Pointers Example

For example, the next program swaps two values of two:

```
void swap (int *a, int *b);
int main()
{
  int m = 25;
  int n = 100;
  printf("m is %d, n is %d\n", m, n);
  swap(&m, &n);
  printf("m is %d, n is %d\n", m, n);
  return 0;
}
void swap (int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Output:

```
m is 25, n is 100
m is 100, n is 25
```

5 Functions

A function is a block of statements that performs a specific task. Suppose you are building an application in C language and in one of your program, you need to perform a same task more than once.

"A function is a block of code that performs a particular task."

There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.

In such case you have two options :

(a) Use the same set of statements every time you want to perform the task

(b) Create a function to perform that task, and just call it every time you need to perform that task.

Using option (b) is a good practice and a good programmer always uses functions while writing codes in C.

Syntax of a function :

```
return_type function_name (argument list)
{
Set of statements - Block of code
}
```

return_type : Return type can be of any data type such as int, double, char, void, short etc. Don't worry you will understand these terms better once you go through the examples below.

function_name : It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing it's name.

argument list : Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example - A function which is used to add two integer variables, will be having two integer argument.

Block of code : Set of C statements, which will be executed whenever a call will be made to the function.

Let's take an example : Suppose you want to create a function to add two integer variables.

Let's split the problem so that it would be easy to understand :

Function will add the two numbers so it should have some meaningful name like sum, addition, etc.

For example lets take the name addition for this function.

```
return_type addition(argument list);
```

This function addition adds two integer variables, which means I need two integer variable as input, lets provide two integer parameters in the function signature. The function signature would be -

return_type addition(int num1, int num2);

The result of the sum of two integers would be integer only. Hence function should return an integer value - **I got my return type** - It would be integer -

int addition(int num1, int num2);

Types of Functions

(1) Predefined standard library functions :

- Library functions in C language are inbuilt functions which are grouped together and placed in a common place called library.
- Each library function in C performs specific operation.
- We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.
- These library functions are created by the persons who designed and created C compilers.
- All C standard library functions are declared in many header files which are saved as file_name.h.
- Actually, function declaration, definition for macros are given in all header files.
- We are including these header files in our C program using "#include<file_name.h>" command to make use of the functions those are declared in the header files.
- When we include header files in our C program using "#include<filename.h>" command, all C code of the header files are included in C program. Then, this C program is compiled by compiler and executed.

Such as puts(), gets(), printf(), scanf() etc - These are the functions which already have a definition in header files (.h files like stdio.h), so we just call them whenever there is a need to use them.

(2) User Defined functions : A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Syntax :

```
Return type   name_of_function    (type 1 arg 1, type2 arg2, typeS arg3)
Return type   function name       argument list of the above syntax
```

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

Function declaration

Function definition

Function call

These three things are represented like

```
int function(int, int, int);    /*function declaration*/
 main()                          /* calling function*/
 {
function (arg1, arg2, arg3);
 }
 int function(type1 arg1,type2 arg2,type3 arg3) /*function definition/*
{
```



```
Local variable declaration;
Statement;
return value;
}
```

Function declaration

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

Function definition

Function definition consists of the whole description and code of the function. It tells about what function is doing what are its inputs and what are its out put It consists of two parts function header and function body.

Syntax :

```
return type function(type1 arg1, type2 arg2, type3 arg3) /*function header*/
{
Local variable declaration;
Statement 1;
Statement 2;
return value
}
```

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

The local variable declared inside a function is local to that function only. It can't be used anywhere in the program and its existence is only within this function. The arguments of the function definition are known as formal arguments.

Function Call

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

Example : function(arg1, arg2, arg3);

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

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

Type of User-defined Functions

There can be 4 different types of user-defined functions, they are :

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
int main()
#include<stdio.h>
void greatNum();        //function declaration
int main()
{
  greatNum();           //function call
  return 0;
}
void greatNum()         //function definition
{
  int i, j;
  printf("Enter 2 numbers that you want to compare…");
  scanf("%d%d", &i, &j);
  if (i > j)
  {
     printf("The greater number is: %d", i);
  }
  else
  {
     printf("The greater number is: %d", j);
  }
}
```

Function with no arguments and a return value

We have modified the above example to make the function greatNum() return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>
int greatNum();              // function declaration
int main()
{
  int result;
  result = greatNum();       // function call
  printf(""The greater number is: %d", result);
  return 0;
}
int greatNum()               // function definition
{
  int i, j, greaterNum;
  printf("Enter 2 numbers that you want to compare...");
  scanf("%d%d", &i, &j);
  if (i > j)
  {
     greaterNum = i;
  }
  else
  {
     greatNum = j;
  }
  // returning the result
  return greaterNum;
}
```


Function with arguments and no return value

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function greatNum() take two int values as arguments, but it will not be returning anything.

```
#include<stdio.h>
void greatNum(int a, int b); //function declaration
int main()
{
  int i, j;
  printf("Enter 2 numbers that you want to compare...");
  scanf("%d%d", &i, &j);
  greatNum{i, j);              // function cal
  return 0;
  }
  void greatNum(int x, int y)    //function definition
  {
  if (x > y)
  {
     printf("The greater number is : %d", x);
  }
  else
  {
     printf("the great number is : %d", y);
  }
}
```

Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>
int greatNum(int a, int b)          // function declaration
int main()
{
  int x, y, result;
  printf("Enter 2 numbers that you want to compare...");
  scanf("%d%d", &i, &i);
  result = greatNum(i, i); // function call
  printf("The greater number is: %d", result);
  return 0;
}
int greatNum(int a, int b)          // function definition
{
  if (a > b)
{
    return a;
  }
  else
  {
    return b;
  }
}
```

Why we need functions

Functions are used because of following reasons :

(a) To improve the readability of code.

(b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.

(c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.

(d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

How to call a function in C

Consider the following C program

- **Example 1 : Creating a user defined function addition ()**

```
#include <stdio.h>
int addition(int num1, int num2)
{
  int sum;
  /* Arguments are used here*/
  sum = num1 + num2;
  /* Function return type is integer so we are returning
  * an integer value, the sum of the passed numbers.
  */
  return sum;
}
int main()
{
  int var1, var2;
  printf("Enter number 1: ");
  scanf("%d",&var1);
  printf("Enter number 2:");
  scanf("%d", &var2);
  /* Calling the function here, the function return type
  * is integer so we need an integer variable to hold the
  * returned value of this function.
  */
  int res = addition(var1, var2);
  printf ("Output: %d", res);
  return 0;
}
```


Output :

Enter number 1: 100

Enter number 2: 120

Output : 220

- **Example 2 : Creating a void user defined function that doesn't return anything**

```
#include <stdio.h>
/* function return type is void and it doesn't have parameters*/
void introduction()
{
  printf("Hi\n");
  printf("My name is kavita \n");
  printf("How are you?");
  /* There is no return statement inside this function, since its
  * return type is void
  */
}
int main()
{
  /*calling function*/
  introduction();
  return 0;
}
```

Output :

```
Hi
My name is kavita
How are you?
```

Nesting of Functions

C language also allows nesting of functions *i.e* to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```
function1()
{
  // function1 body here
  function2();
     //function1 body here
}
```

If function2 also has a called for function1 inside it,then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.

Not able to understand? Lets consider that inside the main() function, function1 is called and its execution starts, then inside function1, we have a call for function2, so the control of program will go to the function2. But as function2 also has a call to function1 in its body, it will call function1, which will again call function2, and this will go on for infinite times, until you forcefully exit from program execution.

Recursion : Recursion is a special way of nesting functions, where a function calls itself inside it. We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```
function1()
{
// function1 body
function1();
// function1 body
}
```

- **Example :** factorial of a number using Recursion

```
#include<stdio.h>
int factorial(int x)        // declaring the function
void main()
{
  int a, b:
  printf("Enter a number...");
  scanf("%d". &a):
  b = factorial(a):         //calling the function named factorial
  printf("%d", b):
}
int factorial(int x)  // defining the function
```



```
{
  int r = 1;
  if (x == 1)
     return 1;
  else
     r = x*factorial (x – 1); // recursion, since the function calls itself
     return r;
}
```

Similarly, there are many more applications of recursion in C language.

Few Points to Note regarding functions

(1) main() in C program is also a function.
(2) Each C program must have at least one function, which is main().
(3) There is no limit on number of functions; A C program can have any number of functions.
(4) A function can call itself and it is known as "Recursion". I have written a separate guide for it.

Benefits of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.
5. C functions are used to avoid rewriting same logic/code again and again in a program.
6. There is no limit in calling C functions to make use of same functionality wherever re
7. We can call functions any number of times in a program and from any place in a prog
8. A large C program can easily be tracked when it is divided into functions.
9. The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

Parameter : Each function has a name to identify it. After performing a certain task using a function, it can return a value. Some functions do not return any value. The data necessary for the function to perform the task is sent as parameters.

Types of parameter: Parameters can be :

1.Actual parameters

2. Formal Parameters

1. Actual parameters : Arguments which are mentioned in the function call is known as the actual argument. For example:

```
func1(12, 23);
```

here 12 and 23 are actual arguments.

Actual arguments can be constant, variables, expressions etc.

- Example :

```
#include <stdio.h>
void addition (int x, int y)
```

```
{
int addition;
addition = x+y;
printf("%d",addition);
}
void main ()
{
addition (2,3);
addition (4,5);
}
```

According to the above C_program, there is a function named addition. In the main function, the value 2 and 3 are passed to the function addition. This value 2 and 3 are the actual parameters. Those values are passed to the method addition, and the sum of two numbers will display on the screen. Again, in the main program, new two integer values are passed to the addition method. Now the actual parameters are 4 and 5. The summation of 4 and 5 will display on the screen.

2. Formal Parameters : Arguments which are mentioned in the definition of the function is called formal arguments. Formal arguments are very similar to local variables inside the function. Just like local variables, formal arguments are destroyed when the function ends.

```
int factorial(int n)
{
  // write logic here
}
```

Here n is the formal argument.

```
// arguments pass by value
# include <stdio.h>
int add (int a, int b)// Formal parameter.
{
  return( a + b );
}
int main()
{
  int x, y, z;
  x = 5;
  y = 5;
  z = add(x,y); // Actual parameter
return 0;
}
// end of program
```

Things to remember about actual and formal arguments.

1. Order, number, and type of the actual arguments in the function call must match with formal arguments of the function.

2. If there is type mismatch between actual and formal arguments then the compiler will try to convert the type of actual arguments to formal arguments if it is legal, Otherwise, a garbage value will be passed to the formal argument.
3. Changes made in the formal argument do not affect the actual arguments.

Types of Function Calls

Functions are called by their names; we all know that, then what is this tutorial for? Well if the function does not have any arguments, then to call a function you can directly use its name. But for functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are :

1. Call by Value
2. Call by Reference

Call by Value

Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function. Hence, the original values are unchanged only the parameters inside the function changes.

```
#include <stdio.h>
void calc(int x);
int main()
{
  int x = 10;
  calc(x);
  // this will print the value of 'x'
  printf("\n value of x in main is %d", x);
  return 0;
}
void calc(int x)
  // changing the value of 'x'
  x = x + 10;
  printf("value of x in calc function is %d", x);
}
```

value of x in calc function is 20

value of x in main is 10

In this case, the actual variable x is not changed. This is because we are passing the argument by value, hence a copy of x is passed to the function, which is updated during function execution, and that copied value in the function is destroyed when the function ends(goes out of scope). So the variable x inside the main() function is never changed and hence, still holds a value of 10.

But we can change this program to let the function modify the original x variable, by making the function calc() return.a value, and storing that value in x.

```
#include<stdio.h>
int calc(int x):
int main()
```

```
    {
      int x = 10;
      x = calc(x):
      printf("value of x is %d", x);
      return 0:
    }
    int calc(int x)
    {
      x = x + 10;
      return x:
    }
value of x is 20
```

Call by Reference

In call by reference we pass..the address (reference) of a variable as argument to any function. Why we pass the address of any variable as argument, then the function function will have access to our variable, as it now knows where it is stored and hence can easily update its value.

In this case the formal parameters can be taken as a reference of a pointers, (don't worry about pointers, we will soon learn about them), in both the case they will change the values of the original variable.

```
#include <stdio.h>
void calc(int *p);        //function taking pointer as argument
int main()
{
  int x = 10:
  calc(&x);               // passing address of 'x' as argument
  printf("value of x is %d". x):
  return(0);
}
void calc(int *p)   //receiving the address in a reference pointer variable
{
  /*
     changing the value directlv that is
     stored at the address passed
  */
  *p = *p + 10;
  }
value of x is 20
```


Difference between call by value and call by reference

	Call By Value	Call by Reference
1.	While calling a function, we pass values of variables to it. Such functions are known as "Call By Values".	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as "Call By References.
2.	In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
3.	With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	.With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.
	// C program to illustrate call by value #include<stdio.h> // Function Prototype void swapx(int x, int y); // Main function int main() { int a = 10, b = 20; // Pass by Values swapx(&a, &b); printf("a=%d b=%d\n", a, b) return 0; } // Swap function that swaps // two values void swapx(int* x, int* y) { int t; t = x x = y; y = t; printf("x=%d\n", x, y); } **Output :** x = 20 y = 10 a = 10 b = 20	// C program to illustrate Call by Reference #include <stdio.h> // Function Prototype void swapx(int*, int*); // Main function int main() { int a = 10, b = 20; // Pass reference swapx(&a, &b); printf("a=%d b=%d\n", a, b) return 0; } // Function to swap two variables // by references void swapx(int* x, int* y) { int t; t = *x; *x = *y; *y = t; printf("x=%d\n", *x, *y); } **Output :** x = 20 y = 10 a = 20 b = 10

	Thus actual values of a and b remain unchanged even after exchanging the values of x and y.	Thus actual values of a and b get changed after exchanging values of x and y.
4.	In call by values we cannot alter the values of actual variables through function calls.	In call by reference we can alter the values of variables through function calls.
5.	**Values of variables are passes by Simple technique.**	**Pointer variables are necessary to define to store the address values of variables.**
6.	Execution time is more then call by value.	Execution time is less then call by value
7.	Call by value is get supported by languages such as : C++.PHP. Visual Basic NET, and C#.	Call by reference is primarily get supported by JAVA.

Strings

In C programming, a string is a sequence of characters terminated with a null character \0. For example:

```
char c[] = "c string";
```

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

c		s	t	r	i	n	g	\0

How to declare a string

Here's how you can declare strings:

```
chars[5]; s[0]
```

s[0]	s[1]	s[2]	s[3]	s[4]

Here, we have declared a string of 5 characters.

How to initialize strings

You can initialize strings in a number of ways.

```
char c[] = "abed";
char c[50] = "abed";
charc[] = {'a', 'b', 'c', 'd', '/0'};
```

c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

Let's take another example:

```
charc[5] = "abcde";
```

Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters. This is bad and you should never do this.

Read String from the user

You can use the scanf() function to read a string.

The scanf() function reads the sequence of characters until it encounters **whitespace** (space, newline, tab etc.).

- **Example 1: scanf() to read a string**

```
tfinclude <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
   return 0;
}
```

Output :

```
Enter name : Dennis Ritchie
Your name is Dennis.
```

Even though Dennis Ritchie was entered in the above program, only "Ritchie" was stored in the name string. It's because there was a space after Dennis.

How to read a line of text

You can use the fgets() function to read a line of string. And, you can use puts() to display the string.

- **Example 2 : fgetsQ and puts()**

```
#include <stdio.h>
int main()
{
   char name[30];
   printf("Enter name:");
   fgets(name, sizeof(name), stdin); // read string
   printf("Name:");
   puts(name); // display string
   return 0;
}
```

Output :

```
Enter name : Tom Hanks
Name : Tom Hanks
```

Here, we have used fgets() function to read a string from the user.

```
fgets(name, sizeof(name), stdlin); // read string
```

The sizeof(name) results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the name string.

To print the string, we have used puts(name);.

- **Note :** The gets() function can also be to take input from the user. However, it is removed from the C standard.

It's because gets() allows you to input any length of characters. Hence, there might be a buffer overflow.

Passing Strings to Functions

Strings can be passed to a function in a similar way as arrays. Learn more about **passing arrays to a function.**

- **Example 3 : Passing string to a Function**

```
#include <stdio.h>
void displayString(char str[]);
int main()
{
   char str[50];
   printf("Enter string:");
   fgets(str, sizeof(str), stdin);
   displayString(str);    // Passing string to a function,
   return 0;
}
void displayString(char str[])
{
   printf("String Output:");
   puts(str);
}
```

String function - strlen

Syntax :

size_t strlen(const char *str)

size_t represents unsigned short

It returns the length of the string without including end character **(terminating char '\0').**

- **Example of strlen :**

```
#include <stdio.h>
#include <string.h>
int main()
{
   charstrl[20] = "BeginnersBook";
   printf("Length of string str1: %d", strlen(str1));
   return 0;
}
```

Output :

Length of string str1: 13

strlen vs sizeof

strlen returns you the length of the string stored in array, however sizeof returns the total allocated size assigned to the array. So if I consider the above example again then the following statements would return the below values.

strlen(strl) returned value 13.

sizeof(strl) would return value 20 as the array size is 20 (see the first statement in main function).

String function strnlen()

Syntax :

size_t strnlen(const char *str, size_t maxlen)

size_t represents unsigned short

It returns length of the string if it is less than the value specified for maxlen (maximum length) otherwise it returns maxlen value.

- **Example of strnlen:**

```
#include <stdio.h>
#include <string.h>
int main()
{
   charstrl[20] = "BeginnersBook";
   printf("Length of string strl when maxlen is 30: %d", strnlen(strl, 30));
   printf("Length of string strl when maxlen is 10: %d", strnlen(strl, 10));
   return 0;
}
```

Output :

Length of string str1 when maxlen is 30:13

Length of string str1 when maxlen is 10:10

Have you noticed the output of second printf statement, even though the string length was 13 it returned only 10 because the maxlen was 10.

String function - strcmp

int strcmp(const char *str1, const char *str2)

It compares the two strings and returns an integer value. If both the strings are same (equal) then this function would return 0 otherwise it may return a negative or positive value based on the comparison.

If string1 < string2 OR stringl is a substring of string2 then it would result in a negative value. If string1 > string2 then it would return positive value.

If string1 == string2 then you would get O(zero) when you use this function for compare strings.

- **Example of strcmp:**

```
#include <stdio.h>
#include <string.h>
int main()
{
   char s1[20] = "BeginnersBook";
   char s2[20] = "BeginnersBook.COM";
   if (strcmp(s1,s2) ==0)
   {
```

```
            printf("string1 and string2 are equal");
        }
        else
        {
            printf("string1 and string2 are different");
        }
        return 0;
    }
```

Output :

string1 and string2 are different

String function - strncmp()

int strncmp(const char *str1, const char *str2, size_t n)

size_t is for unassigned short

It compares both the string till n characters or in other words it compares first n characters of both the strings.

- **Example of strncmp :**

```
#include<stdio.h>
#include <string.h>
int main()
{
  char s1[20] = "BeginnersBook";
  vchars2[20] = "BeginnersBook.COM";
  /* below it is comparing first 8 characters of s1 and s2*/
  if (strncmp(s1,s2, 8) ==0)
  {
     printf("string1 and string2 are equal");
  }
  else
  {
     printf("string1 and string2 are different");
  }
  return 0;
}
```

Output :

string1 and string2 are equal

String function - strcat

char *strcat(char *str1, char *str2)

It concatenates two strings and returns the concatenated string.

- **Example of strcat :**

```
#include <stdio.h>
#include <string.h>
int main()
{
  char s1[10] = "Hello";
  char s2[10] = "World";
  strcat(s1/s2);
     printf("Output string after concatenation: %s", s1);
  return 0;
}
```

Output :

Output string after concatenation: Hello World

String function - strncat

char *strncat(char *str1, char *str2, int n)

It concatenates n characters of str2 to string str1. A terminator char ('\0') will always be appended at the end of the concatenated string.

- **Example of strncat**

```
#include<stdio.h>
#include <string.h>
int main()
{
  char s1[10] = "Hello";
  char s2[10] = "World";
  strncat(s1,s2, 3);
  printf("Concatenation using strncat: %s", s1);
  return 0;
}
```

Output :

Concatenation using strncat: HelloWor

String function - strcpy

char *strcpy(char *str1, char *str2)

It copies the string str2 into string str1, including the end character (terminator char '\0').

- **Example of strcpy**

```
#include <stdio.h>
#include <string.h>
int main()
{
  char sl[30] = "string 1";
  char s2[30] = "string 2 : I'm gonna copied into s1";
```

```
    /* this function has copied s2 into s1*/
    strcpy(s1, s2);
    printf("String s1 is: %s", s1);
    return 0;
}
```

Output :

String s1 is: string 2: I'm gonna copied into s1i

String function - strncpy

```
char *strncpy( char *str1, char *str2, size_t n)
```

size_t is unassigned short and n is a number.

Case1 : If length of str2 > n then it just copies first n characters of str2 into str1.

Case2 : If length of str2 < n then it copies all the characters of str2 into str1 and appends several terminator chars('\0') to accumulate the length of str1 to make it n.

- **Example of strncpy:**

```
#include<stdio.h>
#include<string.h>
int main()
{
  char first[30] = "string1";
  char second[30] = "string2: I'm using strncpy now";
  /* this function has copied first 12 chars of s2 into s1*/
  strncpy(s1,s2,12);
  printf("Strings1 is: %s", s1);
  return 0;
}
```

Output :

String s1 is: string2: I'm using st

String function - strchr

```
char *strchr(char *str, int ch)
```

It searches string str for character ch (you may be wondering that in above definition I have given data type of ch as int, don't worry I didn't make any mistake it should be int only. The thing is when we give any character while using strchr then it internally gets converted into integer for better searching.

- **Example of strchr:**

```
#include<stdio.h>
#include <string.h>
int mainQ
{
  char mystr[30] = "I'm an example of function strchr";
  printf ("%s", strchr(mystr, 'f));
```



```
  return 0;
}
```

Output :

f function strchr

String function - Strrchr

```
char *strrchr(char *str, int ch)
```

It is similar to the function strchr, the only difference is that it searches the string in reverse order, now you would have understood why we have extra r in strrchr, yes you guessed it correct, it is for reverse only.

Now let's take the same above example:

```
#include <stdio.h>
#include <string.h>
int main()
{
  char mystr[30] = "I'm an example of function strchr";
  printf ("%s", strrchr(mystr, 'f'));
  return 0;
}
```

Output :

function strchr

String function - strstr()

```
char *strstr(char *str, char *srch_term)
```

It is similar to strchr, except that it searches for string srch_term instead of a single char.

- **Example of strstr:**

```
#include <stdio.h>
#include <string.h>
int main()
{
  char inputstr[70] = "String Function in C at BeginnersBook.COM";
  printf ("Output string is: %s", strstr(inputstr, 'Begi'));
  return 0;
}
```

Output :

Output string is: BeginnersBook.COM

You can also use this function in place of strchr as you are allowed to give single char also in place of search_term string.

Write a program in C to swap two numbers using the function.

```
#include<stdio.h>
void swap(int *,int *);
int main()
```

```
{
  int n1,n2;
  printf("\n\n Function : swap two numbers using function :\n");
  printf("-----------------------------------------------\n");
  printf("Input 1st number : ");
  scanf("%d",&n1);
  printf("Input 2nd number : ");
  scanf("%d",&n2);
  printf("Before swapping: n1 = %d, n2 = %d ",n1,n2);
     //pass the address of both variables to the function.
  swap(&n1,&n2);
  printf("\nAfter swapping: n1 = %d, n2 = %d \n\n",n1,n2);
  return 0;
}
void swap(int *p,int *q)
{
  //p=&n1 so p store the address of n1, so *p store the value of n1
  //q=&n2 so q store the address of n2, so *q store the value of n2
   int tmp;
  tmp = *p; // tmp store the value of n1
  *p=*q;    // *p store the value of *q that is value of n2
  *q=tmp;   // *q store the value of tmp that is the value of n1
}
```

Sample Output:

```
Function : swap two numbers using function :
-------------------------------
Input 1st number : 2
Input 2nd number : 4
Before swapping: n1 = 2, n2 = 4
After swapping: n1 = 4, n2 = 2
```

Write C code that will display the calculator menu.

```
#include <stdio.h>
#include <stdlib.h>
void displaymenu()
{
 printf("================================\n");
 printf("MENU\n");
 printf("================================\n");
 printf("    1.Add\n");
 printf("    2.Subtract \n");
 printf("    3.Multiply \n");
```



```
 printf("    4.Divide \n");
 printf("    5.Modulus \n");
}
int Add(int a,int b)
{
  return(a+b);
}
  Substract(int a, int b)
  {
  return(a-b);
  }
  Multiply(int a, int b)
  {
  return(a*b);
  }
  float Divide(int a,int b)
  {
  return(a/b);
  }
  int Modulus(int a, int b)
  {
  return(a%b);
  }
  main(int argc, char *argv[])
  {
  //show menu
  displaymenu();
  int yourchoice;
  int a;
  int b;
  char confirm;
  do
  {
     printf("Enter your choice(1-5):");
     scanf("%d:",&yourchoice);
     printf("Enter your two integer numbers:");
     scanf("%d %d",&a,&b);
     printf("\n");
     (yourchoice)
     {
        case 1:printf("Result:%d",Add(a,b));
        break;
```

```
            case 2:printf("Result:%d",Substract(a,b));
            break;
            case 3:printf("Result:%d",Multiply(a,b));
            break;
            case 4:printf("Result:%.2f",Divide(a,b));
            break;
            case 5:printf("Result:%d",Modulus(a,b));
            break;
            default:printf("invalid");
        }
        printf("\nPress y or Y to continue:");
        scanf("%s",&confirm);
    }
    while(confirm=='y'||confirm=='Y');
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Factorial of a Number Using Recursion

```
#include<stdio.h>
long int multiplyNumbers(int n);
int main()
{
  int n;
  printf("Enter a positive integer: ");
  scanf("%d",&n);
  printf("Factorial of %d = %ld", n, multiplyNumbers(n));
  return 0;
}
long int multiplyNumbers(int n)
{
  if (n>=1)
  return n*multiplyNumbers(n-1);
  else
  return 1;
}
```

Output

```
Enter a positive integer: 6
Factorial of 6 = 720
```


Program to find sum of digits using recursion

```
/**
 * C program to calculate sum of digits using recursion
 */
#include <stdio.h>
/* Function declaration */
int sumOfDigits(int num);
int main()
{
  int num, sum;
  printf("Enter any number to find sum of digits: ");
  scanf("%d", &num);
  sum = sumOfDigits(num);
  printf("Sum of digits of %d = %d", num, sum);
  return 0;
}
/**
* Recursive function to find sum of digits of a number
*/
int sumOfDigits(int num)
{
  // Base condition
  if(num == 0)
  return 0;
  return ((num % 10) + sumOfDigits(num / 10));
}
```

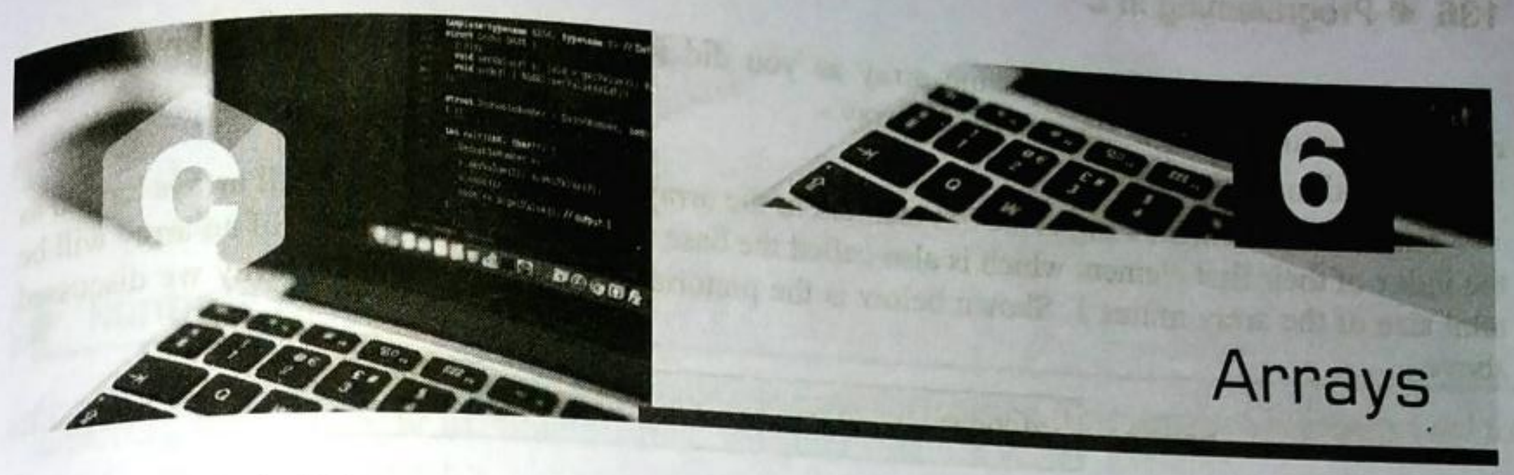
Output:

```
Enter any number to find sum of digits: 1234
Sum of digits of 1234 = 10
```

Program to find reverse of a number using recursion

```
/**
* C program to find reverse of any number using recursion
*/
#include <stdio.h>
#include <math.h>
/* Fuction declaration */
int reverse(int num);
int main()
{
  int num, rev;
```

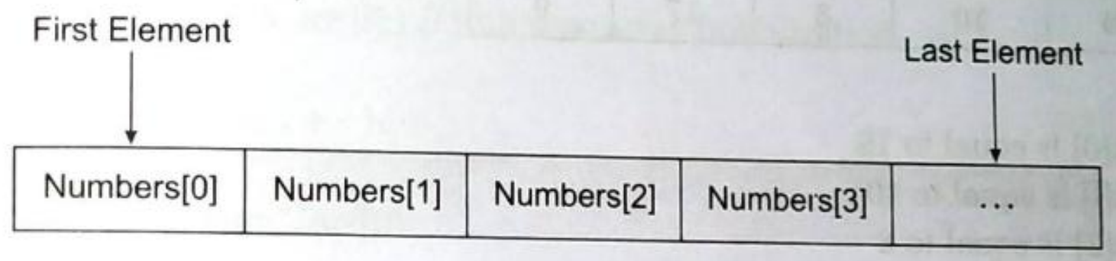
```
        /* Input number from user */
        printf("Enter any number: ");
        scanf("%d", &num);
            /* Call the function to reverse number */
        rev = reverse(num);
        printf("Reverse of %d = %d", num, rev);
        return 0;
    }
    /**
     * Recursive function to find reverse of any number
     */
    int reverse(int num)
    {
      // Find total digits in num
      int digit = (int) log10(num);
      // Base condition
      if(num == 0)
      return 0;
      return ((num%10 * pow(10, digit)) + reverse(num/10));
}
```



Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and … numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the! element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows :

```
datatype arrayName [ arraySize ];
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-etement array called balance of type double, use this statement :

```
double balance[10];
```

Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows :

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created.

Therefore, if you write :

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array -

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above :

balance	1000.0	2.0	3.4	7.0	50.0

Another Example,

```
int mark[5] = {19, 10, 8,17, 9};
```

You can also initialize an array like this.

```
int mark[] = {19,10, 8, 17, 9};
```

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

Here,

mark[0] is equal to 19
mark[l] is equal to 10
mark[2] is equal to 8
mark[3] is equal to 17
mark[4] is equal to 9

Types of initialization

Compile time Array initialization

Compile time initialization of array elements is same as ordinary variable intilization. The general form of initialization of array is,

```
datatype arrayname[size] = {list of values};
/* Here are a few examples */
int marks[4] = {67, 87, 56, 77 };      // integer arrav initialization
float area[5] = {23.4. 6.8. 5.5 };     // float array initialization
int marks[4] = {67, 87, 56, 77, 59 ): // Compile time error
```

One important thing to remember is that when you will give more initialize (array elements) than the declared array size than the compiler will give an error.

```
#include<stdio.h>
void main()
{
  int i;
  int arr[] = {2, 3, 4};  // Compile time array initialization
  for (i = 0; i < 3; i++)
```



```
  {
    printf("d\t", arr[i]);
  }
}
```

2 3 4

Runtime Array Initialization

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large arrays, or to initialize arrays with user specified values. Example,

```
#include<stdio.h>
void main()
{
  int arr[4];
  int i, i;
  printf("Enter array element");
  for (i = 0; i < 4; i ++)
  {
    scanf("%d", & arr [i];) // Run time array initialization
  }
  for (i = 0; i < 4; i ++)
  {
    printf ("%d\n", arr[i]);
  }
}
```

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of element within square brackets after the name of the array. For example -

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays -

```
#include<stdio.h>
int main ()
{
  int n[ 10 ]; /* n is an array of 10 integers */
  int i;
  /* 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 */
  }
```

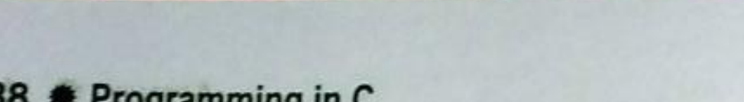
```
    /* output each array element's value */
    for (i = 0; i < 10; i++ )
    {
        printf("Element[%d] = %d\n", i, n[i]);
    }
    return 0;
}
```

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
```

Another example

Suppose you declared an array mark as above. The first element is mark[0], the second element is mark[1] and so on.



Few keynotes :

- Arrays have 0 as the first index, not 1. In this example, mark[0] is the first element.
- If the size of an array is n, to access the last element, the n-1 index is used. In this example, mark[4]
- Suppose the starting address of mark[0] is **2120d**. Then, the address of the mark[1] will be **2124d**. Similarly, the address of mark[2] will be **2128d** and so on.

This is because the size of a float is 4 bytes.

Input and Output Array Elements

Here's how you can take input from the user and store it in an array element.

```
// take input and store it in the 3rd element
scanf("%d", &mark[2]);

// take input and store it in the ith element
scanf("%d", &mark[i-1]);
```

Here's how you can print an individual element of an array.

```
// print the first element of the array
printf("%d", mark[0]);
```



```
// print the third element of the array
printf("%d", mark[2]);

// print ith element of the array
printf("%d", mark[i-1]);
```

- **Example 1: Array Input/Output**

```
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>
int main()
{
   int values[5];
   printf("Enter 5 integers:");
   // taking input and storing it in an array
   for(int i = 0; i < 5; ++i)
   {
       scanf("%d", &values[i]);
   }
   printf("Displaying integers:");
   // printing elements of an array
   for(int i = 0; i < 5; ++i)
   {
       printf("%d\n", values[i]);
   }
   return 0;
}
```

Output :

```
Enter 5 integers: 1
-3
34
0
3
Displaying integers: 1
-3
34
0
3
```

Here, we have used a for loop to take 5 inputs from the user and store them in an array. Then, using another for loop, these elements are displayed on the screen.

- **Example 2: Calculate Average**

```
// Program to find the average of n numbers using arrays
#include <stdio.h>
```

```
int main()
{
  int marks[10], i, n, sum = 0, average;
  printf("Enter number of elements: ");
  scanf("%d", &n);
  for(i=0; i<n; ++i)
  {
     printf("Enter number%d: ",i+l);
     scanf("%d", &marks[i]);
     // adding integers entered by the user to the sum variable
     sum += marks[i];
  }
  average = sum/n;
     printf("Average = %d", average);
     return 0;
}
```

Output :

```
Enter n: 5
Enter number 1 : 45
Enter number 2 : 35
Enter number 3 : 38
Enter number 4 : 31
Enter number 5 : 49
Average = 39
```

Here, we have computed the average of n numbers entered by the user.

Example where arrays are used,

- ❖ to store list of Employee or Student names,
- ❖ to store marks of students,
- ❖ to store list of numbers or characters etc.

Types of Array

1. Single Dimensional Array
2. Two Dimensional Array
3. Three (Multi) Dimensional array

(1) Single Dimensional Array

A dimensional is used representing the elements of the array for example.

```
int a[5];
```

The [] is used for dimensional or the sub-script of the array that is generally used for declaring the elements of the array. For Accessing the Element from the array we can use the Subscript of the Array like this

```
a[3] = 100;
```

This will set the value of 4th element of array.

So there is only the single bracket then it called the Single Dimensional Array.

This is also called as the Single Dimensional Array.

Initialize array at the time of declaration : One way is to initialize one-dimensional array is to initialize it at the time of declaration. You can use this syntax to declare an array at the time of initialization.

```
int a[5] = {10, 20, 30, 40, 50};
```

a[0] is initialized with 10, a[l] is initialized with 20 and so on.

Initialize all elements of an array with 0 (zero) : C program does not assign any default value to declared variables. Same is true for the elements of an array. When we declare an array, all of its elements get initialized with garbage value. If you don't want to get all elements initialized with garbage, you can initialize them with value 0 (zero). You can do this with the help of this syntax.

```
int a[5] = {0};
```

With the following syntax, a[0] to a[5] all are initialized with value 0 (zero).

Important Point: You can initialize an array with 0 (zero) only.

If you want to initialize elements with value 10, this syntax will not work.

```
int a[5] = {10}; //This will not initialize all statements with 10
```

Initialize to define the size of an array : Till now, we declare an array with size. Although, it is also possible to define the size of an array by initializing elements of the array. You can use this syntax for this,

```
int a[] = {10, 20, 30, 40, 50};
```

Remember: if you are not defining size inside [ ] brackets, you must initialize the array (it will define array's size).

If you will not define size of an array inside [ ] brackets, and you are also not initializing the array while declaring. Compiler will show an error "size of array is unknown or zero" and program will not compile.

```
int a[]; //This will cause an error.
```

Initialize array elements individually : Array is a group of variables, each variable is called element of the array. You can initialize all elements separately as you initialize any other variable. See following syntax.

```
int a[5];
a[0] = 10;
a[l] = 20;
a[2] = 30;
a[3] = 40;
a[4] = 50;
```

(2) Two Dimensional Array or the Matrix

The Two Dimensional array is used for representing the elements of the array in the form of the rows and columns and these are used for representing the Matrix A Two Dimensional Array uses the two subscrpts for declaring the elements of the Array.

Like this

```
int a [3] [3];
```

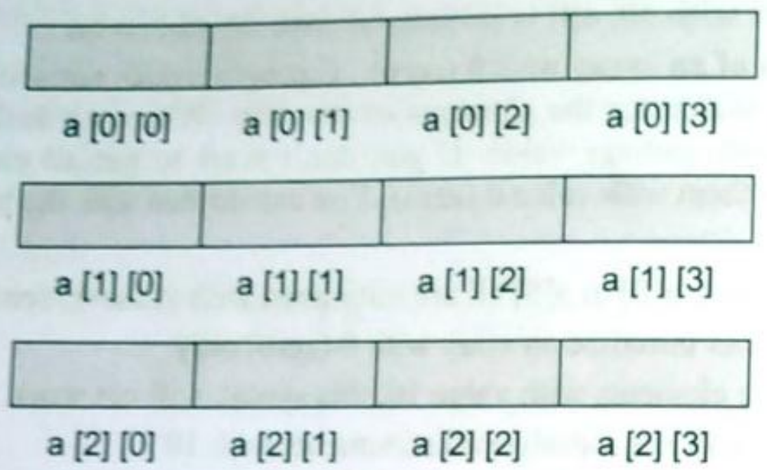
So This is the Example of the Two Dimensional Array in this first 3 represents the total number of Rows and the Second Elements Represents the Total number of Columns. The total Number of

elements are judge by Multiplying the Numbers of Rows * Number of Columns in The Array in the above array the Total Number of elements are 9.

Declaration : Two-dimensional arrays are declared as follows :

data-type array-name[row-size] [column-size]

```
/* Example */
int a [3] [4];
```



An array can also be declared and initialized together. For example,

```
int arr[] [3] = { {0, 0, 0}, {1, 1, 1}};
```

- **Note :** We have not assigned any row value to our array in the above example. It means we can initalize any number of rows. But, we must always specify number of columns, else it will give a compile time error. Here, a 2*3 multi-dimensional matrix is created.

Runtime initialization of a two dimensional :

```
#include<stdio.h>
void main()
{
    int arr [3] [4];
    int i, j, k;
    printf ("Enter array element");
    for (i = 0; i < 3; i++)
    {
        for (i = 0; i < 4; i++)
        {
        scanf("%d", &arr[i] [i]);
        }
}
for (i = 0; i < 3; i++)
{
    for (i = 0; j < 4; i++)
    {
        print("%d", arr[i] [i]);
    }
}
}
```


(3) Multidimensional or the Three Dimensional Array

The Multidimensional Array are used for Representing the Total Number of Tables of Matrix A Three dimensional Array is used when we wants to make the two or more tables of the Matrix Elements for Declaring the Aray Elements we can use the way like this

```
int a[3] [3] [3];
```

In this first 3 represents the total number of Tables and the second 3 represents the total number of rows in the each table and the third 3 represents the total number of Columns in the Tables. So this makes the 3 Tables having the three rows and the three columns.

The Main and very important thing about the array that the elements are stored always in the Contiguous in the memory of the Computer.

Declaration of Multidimensional Array: A multidimensional array is declared using the following syntax :

```
datatype array_name[d1] [d2] [d3] [d4] ..... [dn];
```

Where each d is a dimension, and dn is the size of final dimension.

- **Examples :**

```
1. int table [5] [5] [20];
2. float arr [5] [6] [5] [6] [5];
```

In Example 1 :

- ❖ int designates the array type integer.
- ❖ **table** is the name of our 3D array.
- ❖ Our array can hold 500 integer-type elements. This number is reached by multiplying the value of each dimension. In this case:

  $5 \times 5 \times 20 = 500$.

In Example 2 :

- ❖ Array arr is a five-dimensional array.
- ❖ It can hold 4500 floating-point elements ($5 \times 6 \times 5 \times 6 \times 5 = 4500$).

Can you see the power of declaring an array over variables? When it comes to holding multiple values in C programming, we would need to declare several variables. But a single array can hold thousands of values.

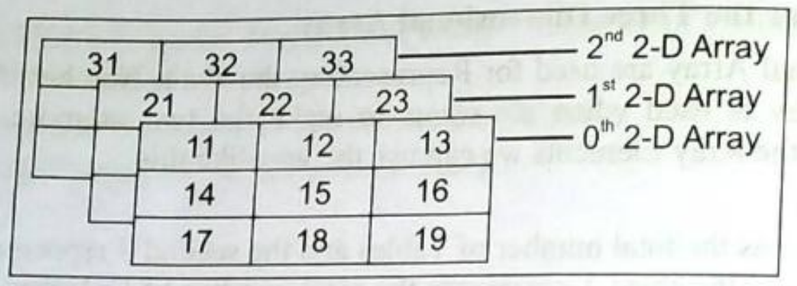
- **Note :** For the sake of simplicity, this tutorial discusses 3D arrays only. Once you grab the logic of how the 3D array works then you can handle 4D arrays and larger.

Explanation of a 30 Array

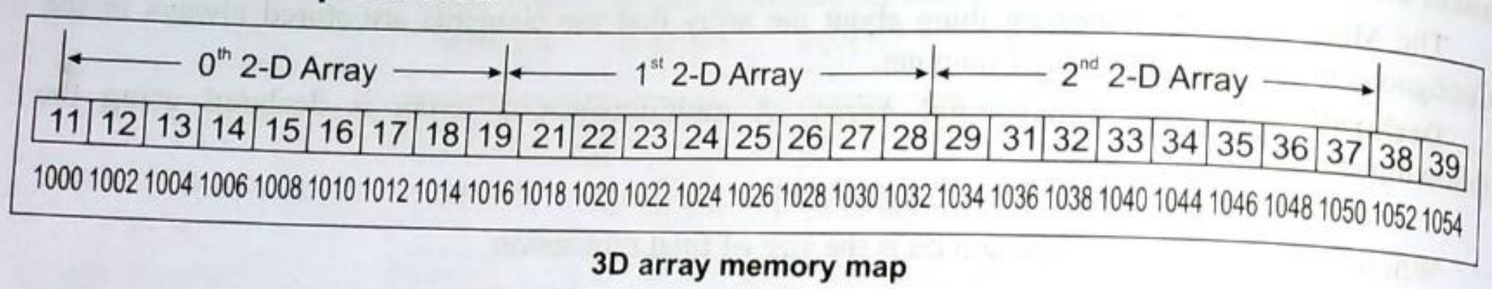
Let's take a closer look at a 3D array. A 3D array is essentially an array of arrays of arrays: it's an array or collection of 2D arrays, and a 2D array is an array of 1D array.

It may sound a bit confusing, but don't worry. As you practice working with multidimensional arrays, you start to grasp the logic.

The diagram below may help you understand :



3D Array Conceptual View



3D array memory map

Initializing a 3D Array : Like any other variable or array, a 3D array can be initialized at the time of compilation. By default, in C, an uninitialized 3D array contains "garbage" values, not valid for the intended use. Let's see a complete example on how to initialize a 3D array:

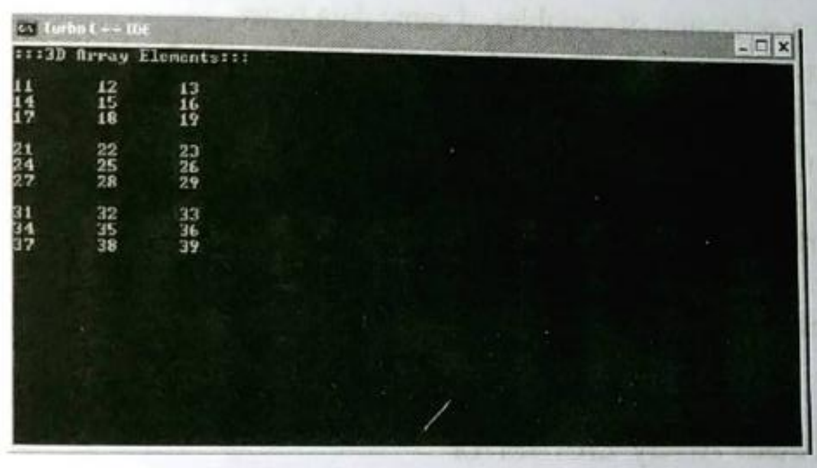
Declaration and Initialization 3D Array

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i, j, k;
int arr[3] [3] [3] =
  {
    {
    {11, 12, 13},
    {14, 15, 16},
    {17, 18, 19}
    },
    {
    {21, 22, 23},
    {24, 25, 26},
    {27, 28, 29}
    },
    {
    {31, 32, 33},
    {34, 35, 36},
    {37, 38, 39}
    },
};
  clrscr();
```



```
printf(":::3D Array Elements:::\n\n");
for(i=0; i<3; i++)
{
    for(j=0; j<3; j++)
    {
    for(k=0;k<3;k++)
    {
    printf("%d\t", arr [i] [j] [k]);
    }
    printf("\n");
  }
  printf("\n");
}
getch();
}
```

Output :



In the code above we have declared a multidimensional integer array named "arr" which can hold 3x3x3 (or 27) elements.

Advantages of Array

- It is better and convenient way of storing the data of same datatype with same size.
- It is used to represent multiple data items of same type by using only single name.
- It allows us to store known number of elements in it.
- It allocates memory in contiguous memory locations for its elements. It does not allocate any extra space/ memory for its elements. Hence there is no memory overflow or shortage of memory in arrays.
- Iterating the arrays using their index is faster compared to any other methods like linked list etc.
- It allows to store the elements in any dimensional array - supports multidimensional array.
- It can be used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.
- 2D arrays are used to represent matrices

Disadvantages of Array

- ❖ We must know in advance that how many elements are to be stored in array.
- ❖ It allows us to enter only fixed number of elements into it. We cannot alter the size of the array once array is declared. Hence if we need to insert more number of records than declared then it is not possible. We should know array size at the compile time itself.
- ❖ Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. And if we allocate less memory than requirement, then it will create problem.
- ❖ The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.
- ❖ Inserting and deleting the records from the array would be costly since we add / delete the elements from the array, we need to manage memory space too.
- ❖ It does not verify the indexes while compiling the array. In case there is any indexes pointed which is more than the dimension specified, then we will get run time errors rather than identifying them at compile time.

Important Points about Arrays in C

- ❖ An array is a collection of variables of same data types.
- ❖ All elements of array are stored in the contiguous memory locations.
- ❖ The size of array must be a constant integral value.
- ❖ Individual elements in an array can be accessed by the name of the array and an integer enclosed in square bracket called subscript/index variable like employee Salary [5].
- ❖ Array is a random access data structure, you can access any element of array in just one statement.
- ❖ The first element in an array is at index 0, whereas the last element is at index (size_of_array -1).

Passing Individual Array Elements

Passing array elements to a function is similar to passing variables to a function.

- **Example 1: Passing an array**

```
#include <stdio.h>
void display(int age1, int age2)
{
   printf("%d\n", age1);
   printf("%d\n", age2);
}
int main()
{
   int ageArray[] = {2, 8, 4, 12};
   // Passing second and third elements to display()
   display(ageArray[1], ageArray[2]);
   return 0;
}
```


Output :

```
8
4
```

- **Example 2 : Passing arrays to functions**

```
// Program to calculate the sum of array elements by passing to a function
#include <stdio.h>
float calculate Sum(float age[]);
int main()
{
   float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};
   // age array is passed to calculate Sum()
   result = calculateSum(age);
   printf("Result = %.2f", result);
   return 0;
}
float calculateSum(float age[])
{
   float sum = 0.0;
   for (int i = 0; i < 6; ++i)
{
      sum += age[i];
}
   return sum;
}
```

Output :

Result = 162. 50

To pass an entire array to a function, only the name of the array is passed as an argument.

```
result = calculate Sum(age);
```

However, notice the use of [] in the function definition.

```
float calculate Sum(float age[])
{
.....
}
```

This informs the compiler that you are passing a one-dimensional array to the function.

Passing Multidimensional Arrays to a Function : To pass multidimensional arrays to a function, only the name of the array is passed to the function(similar to one-dimensional arrays).

- **Example 3 : Passing two-dimensional arrays**

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
   int num[2][2];
```

```
        printf("Enter 4 numbers:\n");
        for (int i = 0; i < 2; ++i)
            for(int j = 0;j<2;++j)
                scanf("%d", &num[i][j]);
        // passing multi-dimensional array to a function
        displayNumbers(num);
        return 0;
    }
    void displayNumbers(int num[2][2])
    {
        printf("Displaying:\n");
        for (int i = 0; i < 2; ++i)
            {
                for (int j = 0;j<2;++j)
                {
                    printf("%d\n", num[i] [j]);
                }
            }
    }
```

Output :

```
Enter 4 numbers :
2
3
4
5
Displaying :
2
3
4
5
```

- **Note :** In C programming, you can pass arrays to functions, however, you cannot return arrays from functions.

Passing array to function in C programming with example : Just like variables, array can also be passed to a function as an argument. We are discussing rn how to pass the array to a function using call by value and call by reference methods.

Passing array to function using call by value method : As we already know in this type of function call, the actual parameter is copied to the formal parameters.

```
#include <stdio.h>
void disp( charch)
{
  printf("%c ", ch);
}
```



```
int main()
{
  char arr[] = {'a','b','c','d', 'e', 'f', 'g','h', 'i','j'};
  for (int x=0; x<10; x++)
  {
    /* I'm passing each element one by one using subscript*/
    disp (arr[x]);
  }
  return 0;
}
```

Output :

```
a b c d e f g h i j
```

Passing array to function using call by reference : When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

```
#include <stdio.h>
void disp( int *num)
{
  printf("%d ", *num);
}
int main()
{
  int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
  for (int i=0; i<10; i++)
  {
    /* Passing addresses of array elements*/
    disp (&arr[i]);
  }
  return 0;
}
```

Output :

```
1 2 3 4 5 6 7 8 9 0
```

Pass an entire array to a function as an argument : In the above example, we have passed the address of each array element one by one using a for loop in C. However you can also pass an entire array to a function like this:

- **Note :** The array name itself is the address of first element of that array. For example if array name is arr then you can say that arr is equivalent to the **&arr[0].**

```
#include <stdio.h>
void myfuncn( int *var1, int var2)
{
    /* The pointer var1 is pointing to the first element of
    * the array and the var2 is the size of the array. In the
```

```
        * loop we are incrementing pointer so that it points to
        * the next element of the array on each increment.
        *
        */
    for(int x=0; x<var2; x++)
    {
      printf("Value of var_arr[%d] is: %d \n", x, *var1);
        /*increment pointer for next element fetch*/
        var1++;
      }
    }
    int main()
    {
      int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
      myfuncn(var_arr, 7);
      return 0;
    }
```

Output :

```
Value of var_arr[0] is : 11
Value of var_arr[1] is : 22
Value of var_arr[2] is : 33
Value of var_arr[3] is : 44
Value of var_arr[4] is : 55
Value of var_arr[5] is : 66
Value of var_arr[6] is : 77
```

Strings and String functions with examples

String is an array of characters. We will see how to compare two strings, concatenate strings, copy one string to another & perform various string manipulation operations. We can perform such operations using the pre-defined functions of "string.h" header file. In order to use these string functions you must include string.h file in your C program.

String Declaration :

Method 1 :

```
char address[] = {'T', 'E', 'X', 'A', 'S', '\0'};
```

Method 2 : The above string can also be defined as -

```
char address[] = "TEXAS";
```

In the above declaration NULL character (\0) will automatically be inserted at the end of the string.

What is Null Char "\0"?

'\0' represents the end of the string. It is also referred as String terminator & Null Character.

String I/O in C programming

Read & write Strings in C using Printf() and Scanf() functions :


```
#include <stdio.h>
#include <string.h>
int main()
{
  /* String Declaration*/
  char nickname[20];
  printf("Enter your Nick name:");
  /* I am reading the input string and storing it in nickname
  * Array name alone works as a base address of array so
  * we can use nickname instead of &nickname here
  */
  scanf("%s", nickname);
  /*Displaying String*/
  printf("%s",nickname);
  return 0;
}
```

Output :

```
Enter your Nick name: Negan
Negan
```

- **Note :** %s format specifier is used for strings input/output

Read & Write Strings in C using gets() and puts() functions :

```
#include<stdio.h>
#include<string.h>
int main()
{
  /* String Declaration*/
  char nickname[20];
  /* Console display using puts */
  puts("Enter your Nick name:");
  /*Input using gets*/
  gets(nickname);
  puts(nickname);
  return 0;
}
```

Write a program in C to read n number of values in an array and display it in reverse order.

```
#include <stdio.h>
void main()
{
int i,n,a[100];
printf("\n\nRead n number of values in an array and display it in reverse order:\n");
```

```
printf("----------\n");
printf("Input the number of elements to store in the array :");
scanf("%d",&n);
printf("Input %d number of elements in the array :\n",n);
for(i=0;i<n;i++)
  {
  printf("element - %d : ",i);
  scanf("%d",&a[i]);
  }
printf("\nThe values store into the array are : \n");
for(i=0;i<n;i++)
  {
  printf("% 5d",a[i]);
  }
printf("\n\nThe values store into the array in reverse are :\n");
for(i=n-1;i>=0;i—)
  {
  printf("% 5d",a[i]);
  }
printf("\n\n");
}
```

Sample Output:

```
Read n number of values in an array and display it in reverse order:
-----------------------------------------------------------------
Input the number of elements to store in the array :3
Input 3 number of elements in the array :
element - 0 : 2
element - 1 : 5
element - 2 : 7
The values store into the array are :
  2   5   7
The values store into the array in reverse are :
  7   5   2
```

Program to count even and odd elements in array

```
/**
* C program to count total number of even and odd elements in an array
*/
#include <stdio.h>
#define MAX_SIZE 100 //Maximum size of the array
int main()
{
```



```
int arr[MAX_SIZE];
int i, size, even, odd;
/* Input size of the array */
printf("Enter size of the array: ");
scanf("%d", &size);
/* Input array elements */
printf("Enter %d elements in array: ", size);
for(i=0; i<size; i++)
{
 scanf("%d", &arr[i]);
}
/* Assuming that there are 0 even and odd elements */
even = 0;
odd  = 0;
for(i=0; i<size; i++)
{
    /* If the current element of array is even then increment even count */
    if(arr[i]%2 == 0)
    {
        even++;
    }
    else
    {
        odd++;
    }
}
  printf("Total even elements: %d\n", even);
  printf("Total odd elements: %d", odd);
  return 0;
}
```

Output:

```
Enter size of the array: 10
Enter 10 elements in array: 5 6 4 12 19 121 1 7 9 63
Total even elements: 3
Total odd element: 7
```

Write a program to print lower triangular matrix and upper triangular matrix of an Array.

```
/* Program to find Lower and Upper Triangle Matrix */
#include<stdio.h>
int main()
```

```
{
  int rows, cols, r, c, matrix[10][10];
  clrscr(); /*Clears the Screen*/
  printf("Please enter the number of rows for the matrix: ");
  scanf("%d", &rows);
  printf("\n");
  printf("Please enter the number of columns for the matrix: ");
  scanf("%d", &cols);
  printf("\n");
  printf("Please enter the elements for the Matrix: \n");
  for(r = 0; r < rows; r++)
  {
     for(c = 0;c < cols;c++)
     {
        scanf("%d", &matrix[r][c]);
     }
  }
  printf("\n The Lower Triangular Matrix is: ");
  for(r = 0; r < rows; r++)
  {
     printf("\n");
     for(c = 0; c < cols; c++)
     {
        if(r >= c)
        {
           printf("%d\t ", matrix[r][c]);
        }
        else
        {
           printf("0");
           printf("\t");
        }
     }
  }
  printf("\n\n The Upper Triangular Matrix is: ");
  for(r = 0; r < rows; r++)
  {
     printf("\n");
     for(c = 0; c < cols; c++)
     {
        if(r > c)
        {
           printf("0");
           printf("\t");
        }
        else
        {
           printf("%d\t ", matrix[r][c]);
        }
     }
  }
  getch();
  return 0;
}
```


C program for matrix addition:

```
#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]);
  printf("Sum of entered matrices:-\n");
  for (c = 0; c < m; c++)
  {
     for (d = 0 ; d < n; d++)
     {
        sum[c][d] = first[c][d] + second[c][d];
        printf("%d\t", sum[c][d]);
     }
     printf("\n");
  }
```

```
    return 0;
}
```

Example: Access members using Pointer

To access members of a structure using pointers, we use the -> operator.

```
#include <stdio.h>
struct person
{
  int age;
  float weight;
};
int main()
{
  struct person *personPtr, person1;
  personPtr = &person1;
  printf("Enter age: ");
  scanf("%d", &personPtr->age);
  printf("Enter weight: ");
  scanf("%f", &personPtr->weight);
  printf("Displaying:\n");
  printf("Age: %d\n", personPtr->age);
  printf("weight: %f", personPtr->weight);
  return 0;
}
```

❑

Structure is a group of variables of different data types represented by a single name. Lets take an example to understand the need of a structure in C programming.

Lets say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when you need to store the data of multiple students then in that case, you would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book -

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows -

```
struct [structure tag]
{
  member definition;
  member definition;
  .....
  member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure -

```
struct Books
{
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
} book;
```

Assign values to structure members

There are three ways to do this.

(1) Using Dot(.) operator

```
var_name.member_name = value;
```

(2) All members assigned in one statement

```
struct struct_name var_name = {value for memeberl, value for memeber2 ...so on for all the members};
```

Example :

```
#include <stdio.h>
/* Created a structure here. The name of the structure is StudentData. */
struct StudentData
{
  char *stu_name;
  int stu_id;
  int stu_age;
};
int main()
{
  /* student is the variable of structure StudentData*/
  struct StudentData student;
  /* Assigning the values of each struct member here */
  student. stu_name = "Steve";
  student.stujd = 1234;
  student. stu_age = 30;
}
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type. The following example shows how to use a structure in a program :

```
#include<stdio.h>
#include <string.h>
struct Books
{
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
};
int main()
{
```



```
  struct Books Book11;   /* Declare Book1 of type Book */
  struct Books Book2;    /* Declare Book2 of type Book */
  /* book 1 specification  */
  strcpy( Bookl.title, "C Programming");
  strcpy( Bookl.author, "Gopal");
  strcpy( Bookl.subject, "C Programming Tutorial");
  Book1.book_id = 221100;
  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;
  /* print Bookl info */
  printf( "Book 1 title : %s\n", Book1.title);
  printf( "Book 1 author: %s\n", Book1.author);
  printf( "Book 1 subject: %s\n", Book1.subject);
  printf( "Book 1 book_id : %d\n", Book1.book_id);
  /* print Book2 info */
  printf( "Book 2 title : %s\n", Book2.title);
  printf( "Book 2 author: %s\n", Book2.author);
  printf( "Book 2 subject: %s\n", Book2.subject);
  printf( "Book 2 book_id : %d\n", Book2.book_id);
  return 0;
}
```

When the above code is compiled and executed, it produces the following result :

```
Book 1 title : C Programming
Book 1 author : Gopal
Book 1 subject : C Programming Tutorial
Book 1 book_id : 221100
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

```
#include <stdio.h>
#indude <string.h>
struct Books
{
char title[50j;
```

```
    char author[50];
    char subject[100];
    int book_id;
};
/* function declaration */
void printBook( struct Books book );
int main()
{
  struct Books Book1;  /* Declare Book1 of type Book */
  struct Books Book2;  /* Declare Book2 of type Book */
  /* book 1 specification */
  strcpy( Book1.title, "C Programming");
  strcpy( Book1.author, "Nuha Ali");
  strcpy( Book1.subject, "C Programming Tutorial");
  Book1.book_id = 6495407;
  /* book 2 specification */
  strcpy( Book2.title, "Telecom Billing");
  strcpy( Book2.author, "Zara Ali");
  strcpy( Book2.subject, "Telecom Billing Tutorial");
  Book2.book_id = 6495700;
  /* print Bookl info */
  printBook( Book1);
  /* Print Book2 info */
  printBook( Book2 );
  return 0;
}
void printBook( struct Books book)
{
  printf( "Book title : %s\n", book.trtle);
  printf("Book author: %s\n", book.author);
  printf( "Book subject: %s\n", book.subject);
  printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result :

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book id : 6495700
```


Nested Structure (Struct Inside Another Struct)

You can use a structure inside another structure, which is fairly possible. Once you declared a structure, the struct struct_name acts as a new data type so you can include it in another struct just like the data type of other data members. Sounds confusing? Don't worry. The following example will clear your doubt.

Example of Nested Structure

Lets say we have two structure like this :

```
Structure1 : stu_address
struct stu_address
{
   int street;
   char *state;
   char *city;
   char *country;
//Structure2: stu_data
struct stu_data
{
   int stu_id;
   int stu_age;
   char *stu_name;
};
   struct stu_address stuAddress;
};
```

As you can see here that I have nested a structure inside another structure.

Assignment for struct inside struct (Nested struct)

Lets take the example of the two structure that we seen above to understand the logic

```
struct stu_data mydata;
mydata.stu_id = 1001;
mydata.stu_age = 30;
mydata. stuAddress. state = "UP"; //Nested struct assignment
```

Access nested structure members

Using chain of "." operator.

Suppose you want to display the city alone from nested struct -

```
printf("%s", mydata. stuAddress. city);
```

Use of Typedef in Structure

typedef makes the code short and improves readability. In the above discussion we have seen that while using structs every time we have to use the lengthy syntax, which makes the code confusing, lengthy complex and less readable. The simple solution to this issue is use of typedef. It is like an alias of struct. Code without typedef

```
struct home_address
{
   int local_street;
   char *town;
   char *my_city;
   char *my_country;
};
struct home_address var;
var.town = "Agra";
//Code using tyepdef
typedef struct home_address
{
   int local_street;
   char *town;
   char *my_city;
   char *my_country;
} addr;
...
...
addr var1;
...
addr var1;
var.town = "Agra";
```

Instead of using the struct home_address every time you need to declare struct variable, you can simply use addr, the typedef that we have defined.

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable :

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows -

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows :

```
struct_pointer->title;
```

Let us re-write the above example using structure pointer.

```
#include<stdio.h>
#include <string.h>
struct Books
{
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
};
```



```
/* function declaration */
void printBook( struct Books *book );
int main( )
{
   struct Books Book1;   /* Declare Bookl of type Book */
   struct Books Book2;  /* Declare Book2 of type Book */
   /* book 1 specification */
   strcpy( Book1. title, "C Programming");
   strcpy( Book1. author, "Nuha AM");
   strcpy( Book1. subject, "C Programming Tutorial");
   Book1.book_id = 6495407;
   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2. author, "Zara Ali");
   strcpy( Book2. subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;
   /* print Book1 info by passing address of Book1 */
   print Book (&Book1 );
   /* print Book2 info by passing address of Book2 */
   print Book ( &Book2 );
   return 0;
}
void printBook( struct Books *book )
{
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

When the above code is compiled and executed, it produces the following result

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Uses of Structures

1. C Structures can be used to store huge data. Structures act as a database.
2. C Structures can be used to send data to the printer.

3. C Structures can in
4. teract with keyboard and mouse to store the data.
5. C Structures can be used in drawing and floppy formatting.
6. C Structures can be used to clear output screen contents.
7. C Structures can be used to check computer's memory size etc.

Array of Structure

Structure is collection of different data type. An object of structure represents a single record in memory, if we want more than one record of structure type, we have to create an array of structure or object. As we know, an array is a collection of similar type, therefore an array can be of structure type.

Syntax for declaring structure array

```
struct struct_name
{
  datatype var1;
  datatype var2;
  ..............
  ..............
datatype varN;
};
struct struct_name obj [ size ];
```

Example for declaring structure array

```
#include<stdio.h>
struct Employee
{
  int Id;
  char Name[25];
  int Age;
  long Salary;
};
void main()
{
  int i;
  struct Employee Emp[ 3 ];          //Statement 1
  for (i=0;i<3;i++)
  {
printf("\nEnter details of %d Employee",i+1);
    printf("\n\tEnter Employee Age :");
    scanf("%d",&Emp[i].Id);
    printf("\n\tEnter Employee Name:");
    scanf("%ld",&Emp[i].Name);
    printf("\n\tEnter Employee Age :");
    scanf("%d", &Emp[i].Age);
```



```
    printf("\n\tEnter Employee Salary :");
    scanf("%ld", &Emp[i].Salary);
  }
  printf("\nDetails of Employees");
  for (i=0;i<3;i++)
  printf("\n%d\t%s\t%d\t%ld",Emp[i].ld,Emp[i].Name,Emp[i].Age,Emp[i].Salary);
}
```

Output :

```
Enter details of 1 Employee
  Enter Employee Id : 101
  Enter Employee Name : Suresh
  Enter Employee Age : 29
  Enter Employee Salary : 45000
Enter details of 2 Employee
  Enter Employee Id : 102
  Enter Employee Name : Mukesh
  Enter Employee Age : 31
  Enter Employee Salary : 51000
Enter details of 3 Employee
  Enter Employee Id : 103
  Enter Employee Name : Ramesh
  Enter Employee Age : 28
  Enter Employee Salary : 47000
Details of Employees
  101    Suresh    29    45000
  102    Mukesh    31    51000
  103    Ramesh    28    47000
```

In the above example, we are getting and displaying the data of 3 employee using array of object. Statement 1 is creating an array of Employee Emp to store the records of 3 employees.

Array within Structure

As we know, structure is collection of different data type. Like normal data type, It can also store an array as well.

Syntax for array within structure

```
struct structname
{
  datatype var1;          //normal variable
  datatype array [size];      // array variable
  ..............
  ..............
  datatype varN;
};
struct structname obj;
```

Example for array within structure

```
struct Student
{
  int Roll;
  char Name[25];
  int Marks[3];          //Statement 1 : array of marks
  int Total;
  float Avg;
};
void main()
{
  int i;
  struct Student S;
  printf("\n\nEnter Student Roll : ");
  scanf("%d",&S.Roll);
  printf("\n\nEnter Student Name : ");
  scanf("%s",&S.Name);
  S.Total = 0;
  for (i=0;i<3;i++)
  {
    printf("\n\nEnter Marks %d : ", i+10);
    scanf("%d",&S.Marks[i]);
    S.Total = S.Total + S.Marks[i];
  }
  S.Avg = S.Total / 3;
  printf("\nRoll : %d",S.Roll);
  printf("\nName : %s",S.Name);
  printf("\nTotal : %d",S.Total);
  printf("\nAverage : %f",S.Avg);
}
```

Output :

```
Enter Student Roll : 10
Enter Student Name : Kumar
Enter Marks 1 : 78
Enter Marks 2 : 89
Enter Marks 3 : 56
Roll : 10
Name : Kumar
Total : 223
Average : 74.00000
```


In the above example, we have created an array Marks[ ] inside structure representing 3 marks of a single student. Marks[ ] is now a member of structure student and to access Marks[ ] we have used dot operator(.) along with object S.

Unions

Like Structures, Union in C Programming is also used to group different data types to organize the data in a structural way. A system reserved keyword union used to create Union.

Unlike Structures, the C union variable will allocate the common memory for all of its union members (*i.e.*, age, name, address, etc.). The size of the allocated common memory is equal to the size of the largest union member. Due to this, we can save much memory.

Due to this common memory, we can't access all the union members at the same time. Because the C union variable holds one member at a time, we can access one union member at a time.

How to define a union?

The basic syntax of the Union in C Programming is as shown below

```
union Union_Name
{
  Data_Type Variable_Name; //Union Member
  Data_Type Variable_Name; //Union Member
  .....
};
```

- **Union :** It is the system reserved keyword used to create union and accessing union members.
- **Union_Name :** Name you want to give for the Union. For example, employees, persons, students.
- **Data_Type :** Data type of the variable that you want to declare. For example, int, float, char, etc.
- **Variable_Name :** Name of the variable. For example, id, age, name, salary.

Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

```
union car
{
  char name[50];
  int price;
};
int main()
{
  union car car1, car2, *car3;
  return 0;
}
```

Another way of creating union variables is:

```
union car
```

```
{
    char name[50];
    int price;
} car1, car2, *car3;
```

In both cases, union variables car1, car2, and a union pointer car3 of union car type are created.

Access members of a union

We use the . operator to access members of a union. To access pointer variables, we use also use the > operator.

In the above example,

- ❖ To access price for car1, car1.price is used.
- ❖ To access price using car3, either (*car3).price or car3->price can be used.

Similarities between Structure and Union

1. Both are user-defined data types used to store data of different types as a single unit.
2. Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
3. Both structures and unions support only assignment = and size of operators. The two structures or unions in the assignment must have the same members and member types.
4. A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
5. '.' operator is used for accessing members.

Difference Between Structure and Union In C

C Structure	C Union
The struct keyword is used to define structure.	The union keyword is used to define structure.
Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space.
Structure occupies higher memory space.	Union occupies lower memory space over structure.
We can access all members of structure at a time.	We can access only one member of union at a time.
Altering the value of a member will not affect other member of structure.	Altering the value of a member will alter other member of structure.
Execution time of structure is fast.	Execution time of union is slow.
Several members of a structure initialized once.	Only first member of union cab be initialized.

Structure example :	Union example:
struct student { int mark; char name[6]; double average; };	union student { int mark; char name[6]; double average; };
For above structure, memory allocation will be like below.	For above union, only 8 bytes of memory will be allocated since double data type will occupy
int mark–2B char name[6] –6B double average –8B Total memory allocation = 2 + 6 + 8 = 16 Bytes	maximum space of memory over other data types. Total memory allocation = 8 Bytes

The following program demonstrates how we can use nested structures.

```
#include<stdio.h>
struct person
{
   char name[20];
   int age;
   char dob[10];
};
struct student
{
   struct person info;
   int roll_no;
   float marks;
};
int main()
{
   struct student s1;
   printf("Details of student: \n\n");
   printf("Enter name: ");
   scanf("%s", s1.info.name);
   printf("Enter age: ");
   scanf("%d", &s1.info.age);
   printf("Enter dob: ");
   scanf("%s", s1.info.dob);
   printf("Enter roll no: ");
   scanf("%d", &s1.roll_no);
```

```
    printf("Enter marks: ");
    scanf("%f", &s1.marks);
    printf("\n*******************************\n\n");
    printf("Name: %s\n", s1.info.name);
    printf("Age: %d\n", s1.info.age);
    printf("DOB: %s\n", s1.info.dob);
    printf("Roll no: %d\n", s1.roll_no);
    printf("Marks: %.2f\n", s1.marks);
    // signal to operating system program ran fine
    return 0;
}
```

Output:

```
Details of student:
Enter name: Phil
Enter age: 27
Enter dob: 23/4/1990
Enter roll no: 78123
Enter marks: 92
*******************************
Name: Phil
Age: 27
DOB: 23/4/1990
Roll no: 78123
Marks: 92.00
```

/* Union in C Programming example */

```
#include <stdio.h>
#include <string.h>
union Employee
{
   int age;
   char Name[50];
   char Department[20];
   float Salary;
};
int main()
{
  union Employee emp1;
  union Employee emp2;
  emp1.age = 28;
  strcpy(emp1.Name, "Chris");
  strcpy(emp1.Department, "Science");
```



```
  emp1.Salary = 32000.70;
  printf("\nDetails of the First Employee \n");
  printf("Employee Age = %d \n", emp1.age);
  printf("Employee Name = %s \n", emp1.Name);
  printf("Employee Department = %s \n", emp1.Department);
  printf("Employee Salary = %.2f \n\n", emp1.Salary);
  printf("Details of the Second Employee \n" );
  emp2.age = 30;
  printf("  Employee Age = %d \n", emp2.age );
  strcpy(emp2.Name, "David");
  printf("  Employee Name = %s \n", emp2.Name );
  strcpy(emp2.Department, "Technology" );
  printf("  Employee Department = %s \n ", emp2.Department );
  emp2.Salary = 35000.20;
  printf("  Employee Salary = %.2f \n ", emp2.Salary );
  return 0;
}
```