# PVKN Govt. College (A), Chittoor

I B.Sc., SEMESTER –I: COMPUTER SCIENCE PAPER – I

# Problem Solving in C

W.E.F. 2021-2022

Subject Code: 21-CSC-1C1 Credits: 04 Teaching Hrs/Week : 4

## SYLLABUS

**Objectives:**

This course aims to provide exposure to problem-solving through programming. It introduces the concepts of the C Programming language.

**Course Learning Outcomes:**

Upon successful completion of the course, a student will be able to:

1. Understand the evolution and functionality of a Digital Computer.
2. Apply logical skills to analyse a given problem
3. Develop an algorithm for solving a given problem.
4. Understand 'C' language constructs, Iterative statements, Array processing, Pointers.
5. Apply 'C' language constructs to solve real time problems.

**UNIT I**

**General Fundamentals:** Introduction to Computers: Block diagram of a Computer, Characteristics and Limitations of Computers – Applications of Computers – Types of Computers – Computer Generations

I/O Devices – Primary, Auxiliary and Cache Memory – Memory Hierarchy – Definition and Types of Software – Definition and Types of Operating System

**UNIT II**

**Introduction to Algorithms, Flowcharts, and Programming Languages**: Algorithms, Key features of Algorithms, – Flow Charts, Symbols used in Flowcharts, Guidelines for developing Flowcharts – Programming Language definition, Generations of Programming Languages.

**Introduction to C:** Introduction – Structure of C Program – Keywords – Identifiers – Basic Data Types in C – Variables – Constants – I/O Statements in C – Operators in C – Programming Examples.

**UNIT III**

**Decision Control and Looping Statements:** Conditional Branching Statements – Iterative Statements – Nested Loops – break and continue Statements – goto Statement.

**Arrays and Strings:** Definition of Array, Declaration of Arrays, Types of Arrays – Operations on Arrays – Declaration of Strings, String handling functions.

**UNIT IV**

**Functions**: Introduction – Function Definition, Implementing User Defined Functions – Scope of variables – Storage Classes – Recursive functions.

**Structure, Union, and Enumerated Data Types:** Structures – Nested Structures – Arrays of Structures – Unions – Arrays of Unions – Enumerated Data Types.

**Pointers:** Introduction to Pointers – declaring Pointer Variables – Null Pointers – Passing Pointers to Functions – Pointers and Arrays.

**UNIT V**

**Files:** Introduction to Files – Opening and Closing a file – Reading Data from Files – Writing Data to Files – Detecting the End-of-file.

**Introduction to C++:** Object Oriented Programming Concepts, Difference between OOP and POP, Basic Structure of a C++ program.

**TEXT BOOKS:**


1. Computer Fundamentals by P.K. Sinha – Sixth Edition, BPB Publications.
2. Yashvant Kanetkar - Let Us ‘C’ – BPB Publications.

**UNIT II**

**Introduction to Algorithms, Flowcharts, and Programming Languages**: Algorithms, Key features of Algorithms, – Flow Charts, Symbols used in Flowcharts, Guidelines for developing Flowcharts – Programming Language definition, Generations of Programming Languages.

**Introduction to C:** Introduction – Structure of C Program – Keywords – Identifiers – Basic Data Types in C – Variables – Constants – I/O Statements in C – Operators in C – Programming Examples.

## Algorithms:

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output.

Algorithms are generally created independent of underlying languages.

### Key Features (or) Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the below mentioned characteristics −

- **Unambiguous −** Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/outputs should be clear and must lead to only one meaning.
- **Input −** An algorithm should have 0 or more well defined inputs.
- **Output −** An algorithm should have 1 or more well defined outputs, and should match the desired output.
- **Finiteness −** Algorithms must terminate after a finite number of steps.
- **Feasibility −** Should be feasible with the available resources.
- **Independent −** An algorithm should have step-by-step directions which should be independent of any programming code.

# Flowcharts

A **flowchart** is a type of diagram that represents a workflow or process.

A **flowchart** can also be defined as a diagrammatic representation of an algorithm, a step-by-step approach to solving a task. The **flowchart** shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows.

A flowchart can be helpful for both writing programs and explaining the program to others.

## Symbols Used In Flowchart

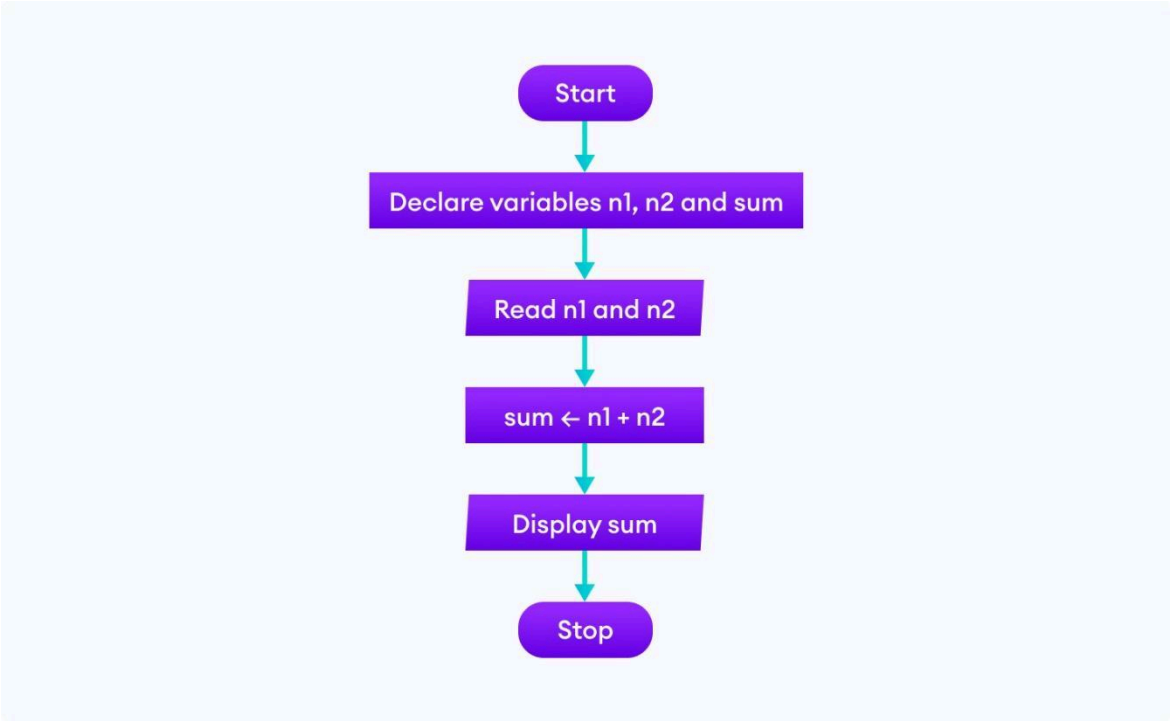
| Symbol | Purpose | Description |
|---|---|---|
|  | Flow line | Indicates the flow of logic by connecting symbols. |
|  | Terminal(Stop/Start) | Represents the start and the end of a flowchart. |
|  | Input/Output | Used for input and output operation. |

| | | |
|---|---|---|
|  | Processing | Used for arithmetic operations and data-manipulations. |
|  | Decision | Used for decision making between two or more alternatives. |
|  | On-page Connector | Used to join different flowline |
|  | Off-page Connector | Used to connect the flowchart portion on a different page. |
|  | Predefined Process/Function | Represents a group of statements performing one processing task. |

---

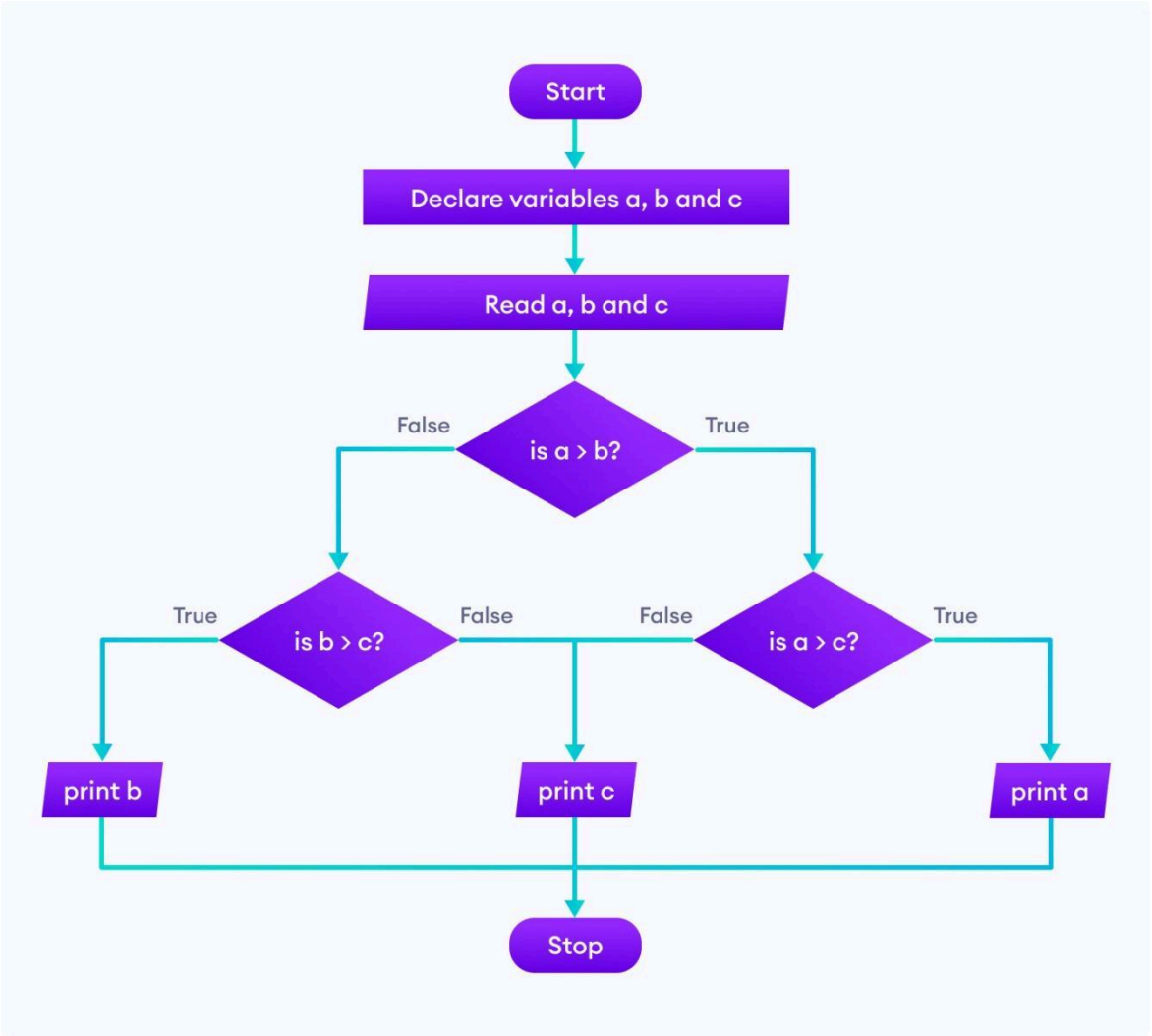
## Examples of flowcharts in programming

**1. Add two numbers entered by the user.**




Flowchart to add two numbers

**2. Find the largest among three different numbers entered by the user.**




## Guidelines for Developing Flowcharts

These are some points to keep in mind while developing a flowchart −

- Flowchart can have only one start and one stop symbol
- On-page connectors are referenced using numbers
- Off-page connectors are referenced using alphabets
- General flow of processes is top to bottom or left to right
- Arrows should not cross each other

## What is a Programming Language?

A programming language is a set of written symbols that instructs the computer hardware to perform specific tasks. Typically, a programming language consists of a vocabulary and a set of rules (called syntax) that the programmer must learn".

## 1st generation of programming languages

- Machine language is the only programming language that the computer can understand directly without translation. It is a language made up of entirely 1s and 0s.
- There is not, however, one universal machine language because the language must be written in accordance with the special characteristics of a given processor. Each type or family of processor requires its own machine language. For this reason, machine language is said to be machine-dependent (also called hardware-dependent).
- In the computer's first generation, programmers had to use machine language because no other option was available.
- Machine language programs have the advantage of very fast execution speeds and efficient use of primary memory. Use of machine language is difficult and time consuming method of programming.
- Machine language is low-level language. Since the programmer must specify every detail of an operation, a low-level language requires that the programmer have detailed knowledge of how the computer works.

## 2nd Generation of programming languages

- The first step in making software development easier and more efficient was the creation of Assembly languages.
- They are also classified as **low-level languages** because detailed knowledge of hardware is still required.
- They were developed in 1950s. Assembly languages use mnemonic operation codes and symbolic addresses in place of 1s and 0s to represent the operation codes.
- A mnemonic is an alphabetical abbreviation used as memory aid. This means a programmer can use abbreviation instead of having to remember lengthy binary instruction codes.
- For example, it is much easier to remember L for Load, A for Add, B for Branch, and C for Compare than the binary equivalents i-e different combinations of 0s and 1s.

## 3rd Generation of programming languages

- Third generation languages, also known as high-level languages, are very much like everyday text and mathematical formulas in appearance.

- They are designed to run on a number of different computers with few or no changes.

### Objectives of high-level languages

- To relieve the programmer of the detailed and tedious task of writing programs in machine language and assembly languages.
- To provide programs that can be used on more than one type of machine with very few changes.
- To allow the programmer more time to focus on understanding the user's needs and designing the software required meeting those needs.
- Most high level languages are considered to be procedure-oriented, or Procedural languages, because the program instructions comprise lists of steps, procedures, that tell the computer not only what to do but how to do it.
- A language translator is required to convert a high-level language program into machine language. Two types of language translators are used with high level languages: compilers and interpreters.

## 4th Generation of programming languages

- Fourth generation languages are also known as very high level languages.
- They are non-procedural languages, so named because they allow programmers and users to specify what the computer is supposed to do without having to specify how the computer is supposed to do it.

### Objectives of fourth generation languages

- Increasing the speed of developing programs.
- Minimizing user effort to obtain information from computer.
- Decreasing the skill level required of users so that they can concentrate on the application rather than the intricacies of coding, and thus solve their own problems without the aid of a professional programmer.
- Minimizing maintenance by reducing errors and making programs that are easy to change.

Five basic types of language tools fall into the fourth generation language category.

1. Query languages
2. Report generators.
3. Applications generators.
4. Decision support systems and financial planning languages.

5. Some microcomputer application software.

## 5th Generation of programming languages

- Natural Languages represent the next step in the development of programming languages, i-e fifth generation languages.
- The text of a natural language statement very closely resembles human speech. In fact, one could word a statement in several ways perhaps even misspelling some words or changing the order of the words and get the same result.
- These languages are also designed to make the computer "smarter". Natural languages already available for microcomputers include Clout, Q&A, and Savvy Retriever (for use with databases) and HAL (Human Access Language).
- The use of natural language touches on expert systems, computerized collection of the knowledge of many human experts in a given field, and artificial intelligence, independently smart computer systems.

**Introduction to C:** Introduction – Structure of C Program – Keywords – Identifiers – Basic Data Types in C – Variables – Constants – I/O Statements in C – Operators in C – Programming Examples.

## INTRODUCTION:

- C is a general-purpose programming language that is extremely popular, simple and flexible.
- It is machine-independent, structured programming language which is used extensively in various applications.
- C was the basics language to write everything from operating systems (Windows and many others) to complex programs like the Oracle database, Git, Python interpreter and more.
- In 1972, a great computer scientist Dennis Ritchie created a new programming language called 'C' at the Bell Laboratories. It was created from 'ALGOL', 'BCPL' and 'B' programming languages. 'C' programming language contains all the features of these languages and many more additional concepts that make it unique from other languages.
- 'C' is a powerful programming language which is strongly associated with the UNIX operating system.

- Even most of the UNIX operating system is coded in 'C'. Initially 'C' programming was limited to the UNIX operating system, but as it started spreading around the world, it became commercial, and many compilers were released for cross-platform systems.

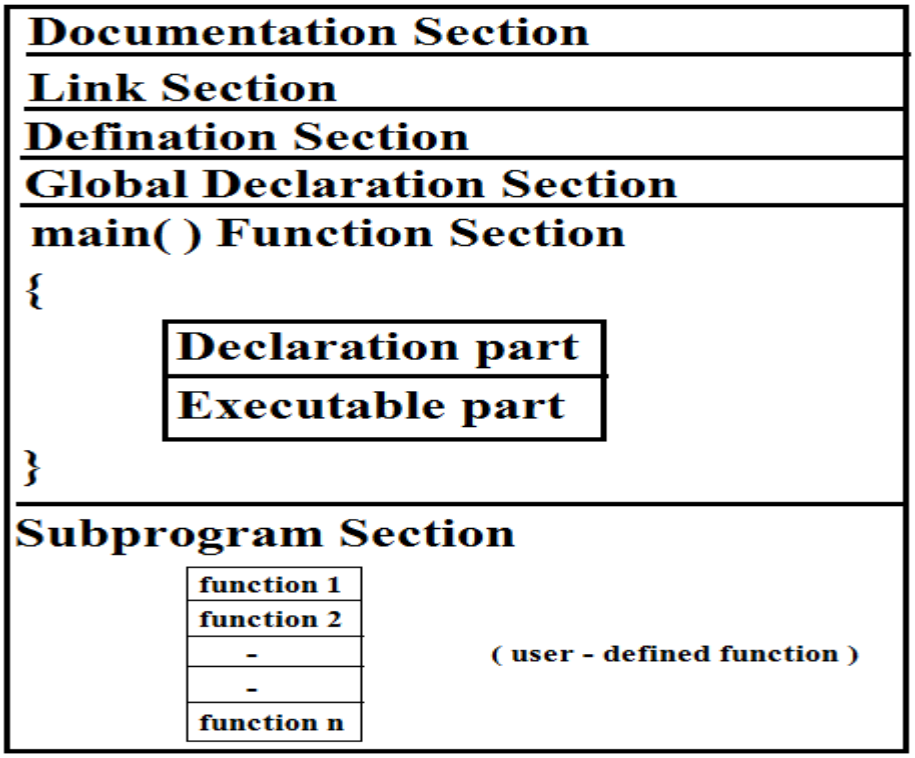
## Where is C used? - Key Applications

1. 'C' language is widely used in embedded systems.
2. It is used for developing system applications.
3. It is widely used for developing desktop applications.
4. Most of the applications by Adobe are developed using 'C' programming language.
5. It is used for developing browsers and their extensions. Google's Chromium is built using 'C' programming language.
6. It is used to develop databases. MySQL is the most popular database software which is built using 'C'.
7. It is used in developing an operating system. Operating systems such as Apple's OS X, Microsoft's Windows, and Symbian are developed using 'C' language. It is used for developing desktop as well as mobile phone's operating system.
8. It is used for compiler production.
9. It is widely used in IOT applications.

## Summary

- 'C' was developed by Dennis Ritchie in 1972.
- It is a robust language.
- It is a low programming level language close to machine language
- It is widely used in the software development field.
- It is a procedure and structure oriented language.
- It has the full support of various operating systems and hardware platforms.
- Many compilers are available for executing programs written in 'C'.
- A compiler compiles the source file and generates an object file.
- A linker links all the object files together and creates one executable file.
- It is highly portable.

## Structure of C Program




Following is the basic structure of a C program.

| | |
|---|---|
| Documentation | Consists of comments, some description of the program, programmer name and any other useful points that can be referenced later. |
| | |
| Link | Provides instruction to the compiler to link function from the library function. |
| | |
| Definition | Consists of symbolic constants. |
| | |
| Global declaration | Consists of function declaration and global variables. |
| | |
| main( )<br>{ | Every C program must have a `main()` function which is the starting point of the program execution. |

| | |
|---|---|
| //Local Declaration<br><br>//Program Statements<br>} | |
| | |
| Subprograms | User defined functions. |
| | |

Lets explore the sections with an example.

## Write a program to print area of a circle.

In the following example we will find the area of a circle for a given radius 10cm.

### Formula

The formula to compute the area of a circle is $\pi r^2$ where $\pi$ is PI = 3.1416 (approx.) and $r$ is the radius of the circle.

Lets write the C code to compute the area of the circle.

```
/**
 * file: circle.c
 * author: ismail
 * date: 2021-02-21
 * description: program to find the area of a circle
 *              using the radius r
 */

#include <stdio.h>

#define  PI  3.1416

float area(float r);

int main(void)
{
  float r = 10;
  printf("Area: %.2f", area(r));
  return 0;
```

```
}

float area(float r) {
    return PI * r * r;
}
```

The above code will give the following output.

```
Area: 314.160000
```

## Different sections of the above code

### Documentation

This section contains a multi line comment describing the code.

```
/**
 * file: circle.c
 * author: ismail
 * date: 2021-02-21
 * description: program to find the area of a circle
 *              using the radius r
 */
```

In C, we can create single line comment using two forward slash `//` and we can create multi line comment using `/* */`.

Comments are ignored by the compiler and is used to write notes and document code.

### Link

This section includes header file.

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

We are including the `stdio.h` input/output header file from the C library.

### Definition

This section contains constant.

```
#define PI 3.1416
```

In the above code we have created a constant PI and assigned 3.1416 to it.

The `#define` is a preprocessor compiler directive which is used to create constants. We generally use uppercase letters to create constants.

The `#define` is not a statement and must not end with a `;` semicolon.

### Global declaration

This section contains function declaration.

```
float area(float r);
```

We have declared an `area` function which takes a floating number (i.e., number with decimal parts) as argument and returns floating number.

### main( ) function

This section contains the `main()` function.

```
int main(void)
{
  float r = 10;
  printf("Area: %.2f", area(r));
  return 0;
}
```

This is the `main()` function of the code. Inside this function we have created a floating variable `r` and assigned 10 to it.

Then we have called the `printf()` function. The first argument contains `"Area: %.2f"` which means we will print floating number having only 2 decimal place. In the second argument we are calling the `area()` function and passing the value of `r` to it.

### Subprograms

This section contains a subprogram, an `area()` function that is called from the `main()` function.

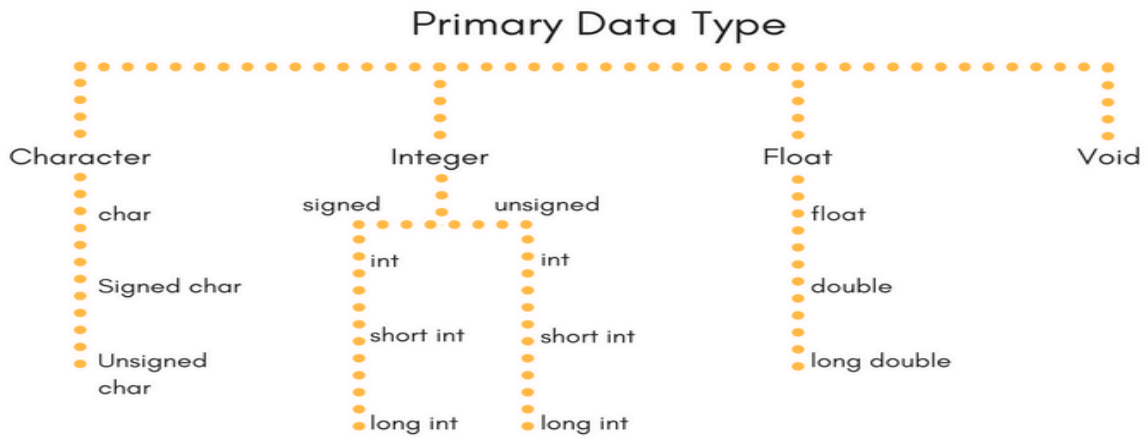
```
float area(float r) {
    return PI * r * r;
}
```

This is the definition of the `area()` function. It receives the value of radius in variable `r` and then returns the area of the circle using the following formula `PI * r * r`.

## Basic Data types in C Language

- Data types specify how we enter data into our programs and what type of data we enter.
- C language has some predefined set of data types to handle various kinds of data that we can use in our program.
- These data types have different storage capacities.
- Data type determines the type of data a variable will hold.
- If a variable `x` is declared as `int`. it means x can hold only integer values.

- Every variable which is used in the program must be declared as what data-type it is.




## Integer type

Integers are used to store whole numbers.(ex: 123, -248, 0, etc.)

**Size and range of Integer type on 16-bit machine:**

| Type | Size(bytes) | Range |
|---|---|---|
| int or signed int | 2 | (-2^15) to (2^15 - 1) |
| unsigned int | 2 | 0 to (2^16 – 1) |
| short int or signed short int | 1 | (-2^7) to (2^7 - 1) |
| unsigned short int | 1 | 0 to (2^8 - 1) |
| long int or signed long int | 4 | (-2^31) to (2^31 - 1) |
| unsigned long int | 4 | 0 to (2^32 – 1) |

---

## Floating point type

Floating types are used to store real numbers.(ex: 12.456, 0.658)

| Type | Size(bytes) |
|---|---|
| float | 4 |
| double | 8 |
| long double | 10 |

---

### Character type

Character types are used to store characters value.(ex: ‘A’, ‘@’)

| Type | Size(bytes) | Range |
|---|---|---|
| char or signed char | 1 | -128 to 127 |
| unsigned char | 1 | 0 to 255 |

---

### void type

`void` type means no value. This is usually used to specify the type of functions which returns nothing.

Ex : void main()

## Variables in C Language

- When we want to store any information (data) on our computer/laptop, we store it in the computer's memory space.
- Instead of remembering the complex address of that memory space where we have stored our data, our operating system provides us with an option to create folders, name them, so that it becomes easier for us to find it and access it.
- Similarly, in C language, when we want to use some data value in our program, we can store it in a memory space and name the memory space so that it becomes easier to access it.
- The naming of an address is known as **variable**.

- Variable is the name of memory location. Unlike constant, variables are changeable; we can change value of a variable during execution of a program. A programmer can choose a meaningful variable name.
- Example: average, height, age, total, etc.

---

**Datatype of Variable:**

A **variable** in C language must be given a type, which defines what type of data the variable will hold.

It can be:

- `char`: Can hold/store a character in it.
- `int`: Used to hold an integer.
- `float`: Used to hold a float value.
- `double`: Used to hold a double value.

---

**Rules to name a Variable:**

1. Variable name must **not** start with a digit.
2. Variable name can consist of alphabets, digits and special symbols like underscore `_`.
3. **Blank or spaces** are **not** allowed in variable name.
4. **Keywords** are **not** allowed as variable name.
5. Upper and lower case names are treated as different, as C is case-sensitive, so it is suggested to keep the variable names in **lower case**.

---

**Declaring, Defining and Initializing a variable**

**Declaration** of variables must be done before they are used in 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 space to the variable.

4. Declaration is more like informing the compiler that there exist a variable with following datatype which is used in the program.

**Defining** a variable means the compiler has to now assign a storage to the variable because it will be used in the program.

**Initializing** a variable means to provide it with a value. A variable can be initialized and defined in a single statement, like:

```
int a = 10;
```

### Tokens in C

Tokens are the smallest elements of a program, which are meaningful to the compiler.

There are 5 types of tokens in C. They are keywords, identifiers, constants, operators, separators.

## Keywords

1. Keywords are those words whose meaning is already defined by Compiler
2. Cannot be used as **Variable Name**
3. There are **32 Keywords** in C
4. C Keywords are also called as **Reserved words** .
5. Example : int, if, else, for, switch, case

Following table represents the keywords in 'C',

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

| | | | |
|---|---|---|---|
| do | if | static | while |

# Identifiers

Identifiers are names given to different entities such as constants, variables, structures, functions etc.

Example

```
int amount;


double totalbalance;
```

In the above example, *amount* and *totalbalance* are identifiers and int, and double are keywords.

### Rules for Naming Identifiers:

- An identifier can only have alphanumeric characters (a-z , A-Z , 0-9) (i.e. letters & digits) and underscore( _ ) symbol.
- Identifier names must be unique
- The first character must be an alphabet or underscore.
- You cannot use a keyword as identifiers.
- Only first thirty-one (31) characters are significant.
- Must not contain white spaces.
- Identifiers are case-sensitive.

## Constants

Constants are the fixed values that never change during the execution of a program. Following are the various types of constants:

### Integer constants

An integer constant is nothing but a value consisting of digits or numbers. These values never change during the execution of a program. Integer constants can be octal, decimal and hexadecimal.

1. Decimal constant contains digits from 0-9 such as,

```
Example, 111, 1234
```

Above are the valid decimal constants.

2. Octal constant contains digits from 0-7, and these types of constants are always preceded by 0.

```
Example, 012, 065
```

Above are the valid octal constants.

3. Hexadecimal constant contains a digit from 0-9 as well as characters from A-F. Hexadecimal constants are always preceded by 0X.

```
Example, 0X2, 0Xbcd
```

Above are the valid hexadecimal constants.

The octal and hexadecimal integer constants are very rarely used in programming with 'C'.

**Character constants**

A character constant contains only a single character enclosed within a single quote ('). We can also represent character constant by providing ASCII value of it.

```
Example, 'A', '9'
```

Above are the examples of valid character constants.

**String constants**

A string constant contains a sequence of characters enclosed within double quotes ("").

```
Example, "Hello", "Programming"
```

These are the examples of valid string constants.

**Real Constants**

Like integer constants that always contains an integer value. 'C' also provides real constants that contain a decimal point or a fraction value. The real constants are also called as floating point constants. The real constant contains a decimal point and a fractional value.

```
Example, 202.15, 300.00
```

# Operators in C

An operator is a symbol that operates on a value or a variable. For example: + is an operator to perform addition.

C has a wide range of operators to perform various operations.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and Decrement Operators
6. Conditional Operator
7. Bitwise Operators
8. Special Operators

| Types of Operators | Description |
|---|---|
| Arithmetic Operators | These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus |
| Assignment Operators | These are used to assign the values for the variables in C programs. |
| Relational Operators | These operators are used to compare the value of two variables. |
| Logical Operators | These operators are used to perform logical operations on the given two variables. |
| Bitwise Operators | These operators are used to perform bit operations on given two variables. |
| Conditional Operator | Conditional operators return one value if condition is true and returns another value is condition is false. |
| Increment and Decrement Operators | These operators are used to either increase or decrease the value of the variable by one. |
| Special Operators | &, *, sizeof( ) |

---

**C Arithmetic Operators**

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

| Operator | Meaning of Operator |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| % | remainder after division (modulo division) |

Ex:

```
int  a=9 , b=4;
```

a+b = 13

a-b = 5

a*b = 36

a/b = 2

a%b=1

**C Relational Operators**

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

| Operator | Meaning of Operator | Example |
|---|---|---|
| == | Equal to | 5 == 3 is evaluated to 0 |
| > | Greater than | 5 > 3 is evaluated to 1 |
| < | Less than | 5 < 3 is evaluated to 0 |
| != | Not equal to | 5 != 3 is evaluated to 1 |
| >= | Greater than or equal to | 5 >= 3 is evaluated to 1 |
| <= | Less than or equal to | 5 <= 3 is evaluated to 0 |

**C Logical Operators**

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

| Operator | Meaning | Example |
|---|---|---|
| && | Logical AND. True only if all operands are true | If c = 5 and d = 2 then, expression ((c==5) && (d>5)) equals to 0. |
| \|\| | Logical OR. True only if either one operand is true | If c = 5 and d = 2 then, expression ((c==5) \|\| (d>5)) equals to 1. |
| ! | Logical NOT. True only if the operand is 0 | If c = 5 then, expression !(c==5) equals to 0. |

**Let a = 5 , b = 5, c= 9**

- (a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true).
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

---

**C Assignment Operators**

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

| Operator | Example | Same as |
|---|---|---|
| = | a = b | a = b |
| += | a += b | a = a+b |
| -= | a -= b | a = a-b |
| *= | a *= b | a = a*b |
| /= | a /= b | a = a/b |
| %= | a %= b | a = a%b |

**C Increment and Decrement Operators**

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Increment/decrement operator are of two types **Postfix** and **Prefix**.

Let a = 10

| | | |
|---|---|---|
| ++ | Increment operator will add 1 to an integer value. | a++ will result to 11<br>++a will result to 11 |
| -- | Decrement operator will subtract 1 from an integer value. | a-- will result to 9<br>--a will result to 9 |

## Conditional Operator

*Conditional operator* is a ternary operator used to evaluate an expression based on some condition. It is a replacement of short if...else statement.

| | | |
|---|---|---|
| ?: | It is used as conditional operator.<br>Syntax of using ternary operator: | `a = 10;`<br>`b = 0;` |

| | | |
|---|---|---|
| | `(condition) ? (true part) :`<br><br>`(false part)` | `b = (a>1) ? a : b;`<br><br>will store the value 10 in `b` as `(a>1)` is true hence true part will execute, assigning the value of `a` in `b`. |

**C Bitwise Operators**

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

| Operators | Meaning of operators | |
|---|---|---|
| & | Bitwise AND | |
| \| | Bitwise OR | |
| ^ | Bitwise exclusive OR | |
| ~ | Bitwise complement | |
| << | Shift left | |
| >> | Shift right | |
| & | *Bitwise AND* performs anding operation on two binary bits value. If both are 1 then will result is 1 otherwise 0. | `0000 1010`<br>`& 0000 0101`<br>`_____`<br>`0000 0000` |
| \| | *Bitwise OR* returns 1 if any of the two binary bits are 1 otherwise 0. | `0000 1010`<br>`\| 0000 0101`<br>`_____`<br>`0000 1111` |
| ^ | Bitwise XOR returns 1 if both the binary bits are different else returns 0. | `0000 1010`<br>`^ 0000 0101`<br>`_____`<br>`0000 1111` |
| ~ | Bitwise COMPLEMENT is a unary operator . It returns the complement of the binary value i.e. if the binary bit is 0 returns 1 else returns 0. | `~ 0000 1010`<br>`_____`<br>`1111 0101` |
| << | *Bitwise LEFT SHIFT* operator is unary operator. It shift the binary bits to the left. It inserts a 0 bit value to the extreme right of the binary value. | `0000 1010 << 2`<br>`= 0010 1000` |

| | | |
|---|---|---|
| >> | *Bitwise RIGHT SHIFT* operator is unary operator. It shifts the binary bits to the right. It inserts a 0 bit value to the extreme left of the binary value. | 0000 1010 << 2<br>= 0000 0010 |

## Special operators

| Operator | Description | Example |
|---|---|---|
| sizeof | Returns the size of an variable | **sizeof(x)** return size of the variable **x** |
| & | Returns the address of an variable | **&x ;** return address of the variable **x** |
| * | Pointer to a variable | ***x ;** will be pointer to a variable **x** |

## Separators: (Additional Input)

**What is separator in C?**
Separator is used to separate tokens. Below are the various separators available in c language.

***; , . : () [] {}***

**Note:** White space is also a separator, but it is not a token.

*Separators* are used to separate one programming element from other. Such as separating keyword from keyword, keyword from identifier, identifier from other identifier etc. They are similar to punctuation marks in English paragraphs.

In C programming every expression is separated using white space character/s, statements are separated from other using semicolon ;.

We can use any number of white space characters to separate two expressions. However we must use at least single white space character to separate one programming element from other.

**Note:** We can write entire C program in two lines if proper separators used. Take an example of below two programs.

```
#include <stdio.h>

int main(){int a=10;int b=20;int c=a+b;printf("Sum=%d",c);return 0;}
```

The above program displays sum of two numbers 10 and 20 using minimum number of separators. However, it is less readable and considered as poor programming practice. We must use proper separators (spaces and indention) to make the program readable.

Consider the same program written with proper separators and is much more readable than previous program.

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

```
int main()
{
	int a = 10;
	int b = 20;
	int c = a + b;

	printf("Sum=%d", c);

	return 0;

}
```

## 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 constants 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 general 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 **% or \.**
- 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;
char grade;
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.

**v) 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 getche(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 devices. 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);
}
```

## Type Casting and Conversion in C (Additional Input)

In a programming language, the expression contains data values of same datatype or different data types. When the expression contains similar datatype values then it is evaluated without any problem. But if the expression contains two or more different datatype values then they must be converted to the single datatype of destination datatype.

Here, the destination is the location where the final result of that expression is stored. For example, the multiplication of an integer data value with the float data value and storing the result into a float variable. In this case, the integer value must be converted to float value so that the final result is a float datatype. value.
In a c programming language, the data conversion is performed in two different methods as follows...

1. Type Conversion
2. Type Casting

### Type Conversion

The type conversion is the process of converting a data value from one datatype to another datatype automatically by the compiler. Sometimes type conversion is also called as **implicit type conversion**. The implicit type conversion is automatically performed by the compiler.
For example, in c programming language, when we assign an integer value to a float variable the integer value automically gets converted to float value by adding decimal value 0. And when a float value is assigned to an integer variable the float value automatically gets converted to integer value by removing the decimal value. To understand more about type conversion observe the following...

```
int i = 10 ;
float x = 15.5 ;
char ch = 'A' ;

i = x ; =======> x value 15.5 is converted as 15 and assigned to variable i

x = i ; =======> Here i value 10 is converted as 10.000000 and assigned to variable x

i = ch ; =======> Here the ASCII value of A (65) is assigned to i
```

### Type Casting

Type casting is also called as **explicit type conversion**. Compiler converts data from one datatype to another datatype implicitly. When compiler converts implicitly, there may be a data loss.In such case, we convert the data from one datatype to another datatype using explicit type conversion. To perform this we use the **unary cast operator**. To convert data from one type to another type we specify the target datatype

in paranthesis as a prefix to the data value that has to be converted. The general syntax of type casting is as follows...

**(TargetDatatype) DataValue**

### Example

int totalMarks = 450, maxMarks = 600 ;
float average ;

**average = (float) totalMarks / maxMarks * 100 ;**

In the above example code, both totalMarks and maxMarks are integer data values. When we perform totalMarks / maxMarks the result is a float value, but the destination (average) datatype is float. So we use type casting to convert totalMarks and maxMarks into float datatype.

### Example Program

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

void main(){
   int a, b, c ;
   float avg ;
   printf("Enter any three integer values : ") ;
   scanf("%d%d%d", &a, &b, &c) ;

   avg = (a + b + c) / 3 ;
   printf("avg before casting = %f\n",avg);

   avg = (float)(a + b + c) / 3 ;
   printf("avg after casting = %f\n",avg);
}
```

**Output:**

```
Enter any three integer values : 5 10 20

avg before casting = 11.000000

avg after casting = 11.666666
```