C for Statistics

Computer Programming

Computer programming is a process of formulating a computing problem to executable computer programs. Programming involves activities such as analysis, generating algorithms and implementation of algorithms in a target programming language.

Programming Language

A ‘Programming Language’ is a formal language that specifies a set of instructions that can be used to produce various kinds of output. Programming languages generally consist of instructions for a computer. Programming languages can be used to create programs that implement specific algorithms.

Basically, programming languages can be divided into the following two categories according to how the computer understands them.

1. Low-level language
2. High-level language

Low-level language

Low-level computer languages are either machine languages or languages very close them. A computer cannot understand instructions given to it in high-level languages or in English. It can only understand and execute instructions given in the form of machine language i.e. binary (0 and 1). There are two types of low-level languages:

1. Machine Language
2. Assembly Language

Machine Language

Machine language is the lowest and most elementary level of programming language. Machine language is basically the only language that a computer can understand. In fact, a manufacturer designs a computer to obey just one language, its machine language, which is represented inside the computer by a string of binary digits (bits) 0 and 1. The symbol 0 stands for the absence of an electric pulse and the 1 stands for the presence of an electric pulse. Since a computer is capable of recognizing electric signals, it understands machine language.

Advantages	**Disadvantages**
1. Machine language makes fast and efficient use of the computer.	1. All operation codes have to be remembered
2. It requires no translator to translate the code. It is directly understood by the computer.	2. All memory addresses have to be remembered.
	3. It is hard to find errors in a program written in the machine language.

Assembly Language

Assembly language is a modified version of machine language in which instructions are given in the form of alphanumeric symbols like ADD, SUM, MOV etc. instead of 0’s and l’s. Because of this feature, assembly language is also known as ‘Symbolic Programming Language.' This language is also very difficult and needs a lot of practice to master it because there is only a little English support in this language. The instructions of the assembly language are converted to machine codes by a language translator (known as ‘Assembler’) and then they are executed by the computer.

Advantages	Disadvantages
1. Assembly language is easier to understand and use as compared to machine language.	1. Like machine language, it is also machine dependent/specific.
2. It is easy to locate and correct errors.	2. Since it is machine dependent, the programmer also needs to understand the hardware.
3. It is easy to modify.	

High-Level Language

High-level computer languages use formats that are similar to English. The purpose of developing high-level languages was to enable people to write programs easily in their own native language environment (English). High-level languages are basically symbolic languages that use English words and mathematical symbols rather than alphanumeric symbols. Each instruction in the high-level language is translated into many machine language instructions by a high-level language translator (‘Compiler’ or ‘Interpreter’).

Advantages	Disadvantages
1. High-level languages are user-friendly.	1. A high-level language has to be translated into the machine language by a translator, which takes up time.
2. They are similar to English and use English vocabulary and well-known symbols.	2. The object code generated by a translator might be inefficient compared to an equivalent assembly language program.
3. They are easier to learn.	
4. They are easier to maintain.	
5. They are problem-oriented rather than machine based.	
6. A program written in a high-level language can be translated into many machine languages and can run on any computer for which there exists an appropriate Compiler / Interpreter.	

Types of High-Level Languages

Many High-level computer languages have been developed for achieving a variety of different tasks. Some are fairly specialized, and others are quite general. These languages, categorized according to their use, are:

1. **Algebraic Formula-Type Processing:** These languages are oriented towards the computational procedures for solving mathematical and statistical problems. Some examples include:
    - BASIC (Full name is “Beginners All Purpose Symbolic Instruction Code”)
    - FORTRAN (Full name is “Formula Translation”)
    - ALGOL (Full name is “Algorithmic Language”)
    - APL (Full name is “A Programming Language”)
2. **Business Data Processing:** These languages emphasize their capabilities for maintaining data processing procedures and files handling problems. Examples are:
    - COBOL (Full name is “Common Business Oriented Language”)
    - RPG (Full name is “Report Program Generator”)
3. **Multipurpose Language:** These languages are useful for algebraic procedures, data processing, string processing and web development. Examples are:
    - C
    - C++
    - Java
    - JavaScript (It is the main client-side web development language for all modern web-browsers like Google Chrome, Mozilla Firefox, Internet Explorer, Opera, etc.)
    - PERL (Full name is “Practical Extraction and Reporting Language”. Suitable for server-side web development and string parsing.)
    - Pascal (After the name of French philosopher and mathematician *Blaise Pascal*).
    - PHP (Full name is “Personal Home Page”. Suitable for server-side web development.)
    - Python (Python's name is derived from the British comedy group “Monty Python”, whom Python creator *Guido van Rossum* enjoyed while developing the language.)
    - Ruby (This name is taken from the name of the gemstone ruby)

Compiler

Compiler is a computer program that translates code written in a high-level programming language (like C, C++, JavaScript or Java) into low-level code directly executable by the computer or another program such as a virtual machine.

For example, the Java compiler converts Java code to Java Bytecode executable by the JVM (Java Virtual Machine). Other examples are V8, the JavaScript engine from Google Chrome which converts JavaScript code to machine code or GCC which can convert code written in programming languages C and C++ to native machine code.

Interpreter

An interpreter is a special kind of program for the conversion of a high-level program statement into machine code just before the program statement is to be executed. During each conversion, the interpreter converts the source code into an intermediate code before processing it into the machine for the generation of machine code. Each part of the code is interpreted line by line and then executed separately in a sequential manner. So, if an error is found in any part of the code, the interpreter will stop the interpretation without converting the next set of codes.

As interpreters check the source code line by line, translation and execution occurs immediately one statement at a time. As a result, they are 2 to 10 times slower than compilers. But this very disadvantage makes interpreters easier to user, especially for beginners; errors once found are immediately displayed and corrected by the user, before the program is executed.

C Language

C Programming is an ANSI/ISO standard and powerful programming language for developing real time applications. C programming language was invented by Dennis Ritchie at the Bell Laboratories in 1972. It was invented for implementing UNIX operating system. C is most widely used programming language even today. C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by following ways:

1. **C as a mother language:** C language is considered as the mother language of all the modern languages because most of the compilers, Java virtual machines, Kernel of operating systems, etc. are written in C language and most of languages follows C syntax e.g. C++, Java etc. It provides the core concepts like array, functions, file handling etc. that is being used in many languages like C++, Java, C# etc.
2. **C as a system programming language:** A system programming language is used to create system softwares. C language is a system programming language because it can be used to do low level programming (e.g. Driver and Kernel). It is generally used to create hardware devices, OS, Drivers, Kernels etc. For example, Linux Kernel is written in C. It can't be used directly in internet programming like Java, .net, PHP, PERL, etc.
3. **C as a procedural language:** A procedure is known as function, method, routine, subroutine etc. A procedural language specifies a series of steps or procedures for the program to solve the problem. A procedural language breaks the program into functions, data structures etc. C is a procedural language. In C, variables and function prototypes must be declared before being used.
4. **C as a structured programming language:** A structured programming language is a subset of procedural language. Structure means to break a program into parts or blocks so that it may be easy to understand. In C language, we break the program into parts using functions. It makes the program easier to understand and modify.
5. **C as a mid-level programming language:** C is considered as a middle-level language because it supports the feature of both low-level and high-level language. C language program is converted into assembly code, supports pointer arithmetic (features of low-level), but it is machine independent (feature of high-level).

The History of the C Language

The ***C*** programming language was devised in the early 1970s by *Dennis M. Ritchie,* an employee from ***Bell Labs***.

In the 1960s Ritchie worked, with several other employees of ***Bell Labs***, on a project called ***Multics***. The goal of the project was to develop an operating system for a large computer that could be used by a thousand users. In 1969 ***Bell Labs*** withdrew the project, because the project could not produce an economically useful system. So the employees of ***Bell Labs*** had to search for another project to work on (mainly *Dennis M. Ritchie* and *Ken Thompson*). At that time a complete Operating System known as ***UNIX*** was born. The whole system was written in ***assembly*** code. Besides ***Assembler*** and ***Fortran***, ***UNIX*** also had an interpreter for the programming language ***B*** which was developed in 1969-70 by *Ken Thompson* based on ***BCPL*** (Basic Combined Programming Language).

In the early days computer code was written in ***assembly*** code. To perform a specific task, one had to write many pages of code. A high-level language like ***B*** made it possible to write the same task in just a few lines of code. So the language ***B*** was used for further development of the ***UNIX*** system because of its efficient and faster coding techniques.

Ken Thompson and Dennis Ritchie

A drawback of the ***B*** language was the absence of ***data-types***. The lack of this thing formed the reason for *Dennis M. Ritchie* to develop the programming language ***C***. So in 1971-73 *Dennis M. Ritchie* turned the ***B*** language into the ***C*** language, keeping most of the language ***B*** syntax while adding ***data-types*** and many other changes. The ***C*** language had a powerful mix of high-level functionality and the detailed features required to program an Operating System. Therefore many of the ***UNIX*** components were eventually rewritten in ***C*** (the ***UNIX*** kernel itself was rewritten in 1973).

The programming language C was written down, by *Kernighan* and *Ritchie*, in their classic book *"The C Programming Language, 1st edition"*. For years this book was the standard on the language ***C***. In 1983 a committee was formed by the American National Standards Institute (***ANSI***) to develop a modern definition for the programming language ***C***. In 1988 they delivered the final standard definition ***ANSI C***. (The standard was based on the book from *"The C Programming Language, 1st edition"*). The standard ***ANSI C*** made little changes on the original design of the ***C*** language. (They had to make sure that old programs still worked with the new standard). Later on, the ***ANSI C*** standard was adopted by the

International Standards Organization (***ISO***). The correct term should therefore be ***ISO C***, but everybody still calls it ***ANSI C***.

C Programs & Compilers

A C program can vary from 3 lines to millions of lines and it should be written into one or more text files with extension ".c"; for example, *hello.c*. We can use any text editor (like Notepad, gedit, vi, etc.) to write our C program ie. “C source code” into a file.

The source code written in text file is the human readable source for our program. It needs to be "compiled" into machine language so that our CPU can actually execute the program as per the instructions given. A compiler compiles the source codes into final executable programs. A list of most frequently used C compilers are given below:

Name	**Author**	**Supporting OS**	**License type**
GCC C/C++	GNU Project	MS Windows, Unix, Linux, Solaries, Mac	GPL
(Borland) Turbo C/C++	Embarcadero Technologies	MS Windows	Proprietary
Visual C++	Microsoft	MS Windows	Freeware
Digital Mars C/C++	Digital Mars	MS Windows	Proprietary
Intel C/C++ Compiler	Intel	MS Windows, Linux, Mac	Proprietary (Freeware for most non-commercial applications)
Xcode	Apple	Mac	Freeware

C Program Structure

A C program basically consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello, World":

```
#include <stdio.h>

int main()
{

      /* my first program in C */

      printf("Hello, World");

      return 0;
}
```

Let us take a look at the various parts of the above program:

- The first line of the program `#include <stdio.h>` is a preprocessor command, which tells a C compiler to include *stdio.h* file before going to actual compilation.
- The next line `int main()` is the main function where the program execution begins.
- The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
- The next line `printf(...)` is another function available in C which causes the message "Hello, World" to be displayed on the screen.
- The next line `return 0` terminates the `main()` function and returns the value 0.

Basic Syntax

Almost every program we write on a computer has different type of format. That format is called as Syntax. A syntax is a set of rules, principles, and processes that govern the structure of a program in any given computer language.

Now let us look at the basic syntax of c programming.

- **Tokens:** A smallest individual unit in C program is known as C Token. Tokens are either keywords, identifiers, constants, variables or any symbol which has some meaning in C language. A C program can also be called as a collection of various tokens. Let us consider the following statement:

  ```
  printf("Hello, World");
  ```

  The tokens in this statement are **printf**, **(**, **"Hello,World"**, **)** and **;**

  So C tokens are basically the building blocks of a C program.

- **Semicolons:** In a C program, the semicolon is used to mark the end of a statement and beginning of another statement. Absence of semicolon at the end of any statement will mislead the compiler to think that this statement is not yet finished and it will add the next consecutive statement after it, which may lead to compilation (syntax) error.

- **Comments:** Comments are plain simple text in C programs that are not compiled by the compiler. We write comments for better understanding of the program. Though writing comments is not compulsory, but it is recommended to make our program more descriptive. It makes the code more readable.

  There are two ways in which we can write comments.
  1. **Using //:** This is used to write a single line comment.
  2. **Using /* */:** The statements enclosed within /* and */, are used to write multiline comments.

  It should be noted that we cannot have comments within comments and they do not occur within a string.

- **Identifiers:** In C language, identifiers are the names given to variables, constants, functions and user-define data. These identifiers are defined against a set of rules given below.
    1. An Identifier can only have alphanumeric characters (a-z , A-Z , 0-9) and underscore (_).
    2. The first character of an identifier can only contain alphabet (a-z , A-Z) or underscore (_).
    3. Identifiers are also case sensitive in C. For example ‘mean’ and ‘Mean’ are two different identifiers in C.
    4. Keywords are not allowed to be used as identifiers.
    5. No special characters, such as semicolon, whitespaces, slash, comma, @, $, #, %, etc. are permitted to be used as identifier.

- **Keywords:** Keywords are preserved words that have special meaning in C language. The meaning of C language keywords has already been described to the C compiler. These meaning cannot be changed. Thus, keywords cannot be used as variable names because that would try to change the existing meaning of the keyword, which is not allowed. There are total 32 keywords in C language.

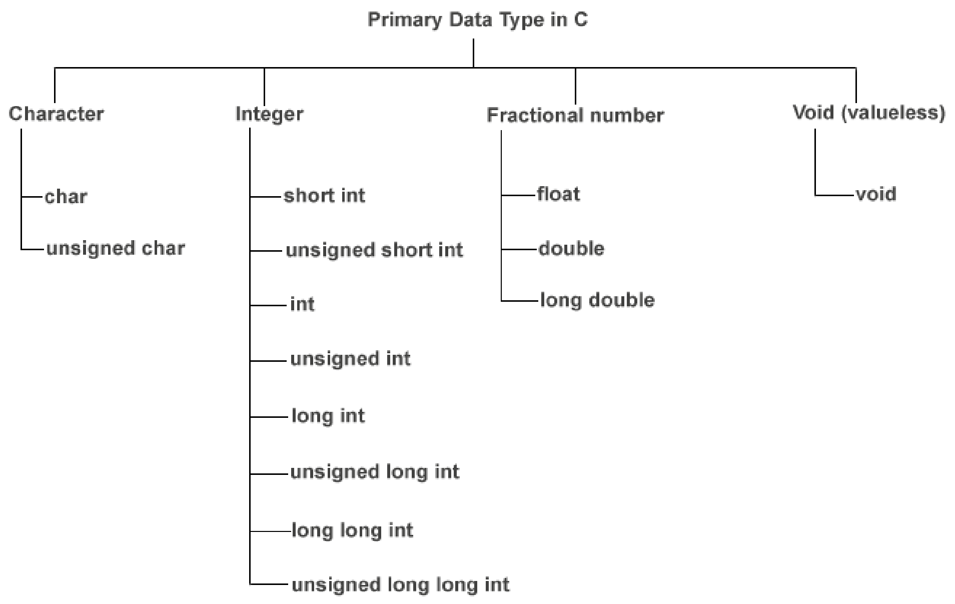
auto	break	case	const	char	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	volatile	void	while

Data Types in C

In C programming, variables and functions should be declared before it can be used. Each variable and function in C has an associated data type. Data types simply refer to the type and size of data associated with variables and functions. Different data types have different ranges upto which they can store numbers. These ranges may vary from compiler to compiler.

C language supports three different types of data types:

1. **Primary data types:** These are fundamental data types in C.



Details of primary data types along with their ranges, memory requirement and format specifiers in a **32 bit gcc compiler** are given below:

Data Type	**Memory (in bytes)**	**Range**	**Format Specifier**	**Description**
`char`	1	-128 to 127	%c	The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
`unsigned char`	1	0 to 255	%c	
`short int`	2	-32,768 to 32,767	%hd	
`unsigned short int`	2	0 to 65,535	%hu	
`int`	4	-2,147,483,648 to 2,147,483,647 ie. -(2^31) to (2^31)-1	%d	As the name suggests, an int variable is used to store an integer.
`unsigned int`	4	0 to 4,294,967,295	%u	
`long int`	4	-2,147,483,648 to 2,147,483,647 ie. -(2^31) to (2^31)-1	%ld	
`unsigned long int`	4	0 to 4,294,967,295 ie. 0 to (2^32)-1	%lu	
`long long int`	8	−9,223,372,036,854,775,807 to 9,223,372,036,854,775,807 ie. -(2^63) to (2^63)-1	%lld	
`unsigned long long int`	8	0 to 18,446,744,073,709,551,615 ie. 0 to (2^64)-1	%llu	
`float`	4	1.18E-38 to 3.4E+38	%f, %g, %G, %e, %E	It is used to store decimal numbers with 6 digits of precision.
`double`	8	2.23E-308 to 1.79E+308	%lf, %lg, %lG, %le, %lE	It is used to store decimal numbers with 15 digits of precision.
`long double`	12	3.4E-4932 to 1.1E+4932	%Lf, %Lg, %LG, %Le, %LE	It is used to store decimal numbers with 18 digits of precision.
`void`				The void type specifies that no value is available. This can be used in functions and pointers.

2. **Derived data types:** Data types that are derived from fundamental data types are called derived data types. Derived data types don't create a new data type but,instead they add some functionality to the basic data types. In C, two derived data type are : Array & Pointer.
    - Array: An array is a collection of variables of same type. They are stored in contagious memory allocation. For example: **`int a[10]; float b[20];`** etc.
    - Pointer: A pointer is a special variable that holds a memory address (location in memory) of another variable. For example: **`int i = 10; int *j; j = &i;`**

3. **User defined data types:** The user defined data types enable a programmer to invent his own data types and define what values it can take on. C supports two types of user defined data types:
    - **Structures:** A structure is a collection of variables, constants and arrays of various data types. The main difference between an array and a structure is that the members of a structure are of different types. This offers excellent flexibility when working with structures. Now, consider the following example:

```
#include <stdio.h>

typedef struct student_records
{
      char *name;
      int age;
      float weight;
      float height;
} student;

int main()
{

      student x;

      x.name = "Mr. X";
      x.age = 22;
      x.weight = 56.6;
      x.height = 5.4;

      return 0;
}
```

Here we define a structure named student with four variables name, age, weight & height. In the main function we create a variable **`x`** of type **`student`** and assign values to its all member variables.

    - **Unions:** A union is a special data type available in C that allows storing different data types in the same memory location. We can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose. The following example shows how to use unions in a program.

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

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

int main()
{
	union Data data;

	data.i = 10;

	printf( "Memory size occupied by data : %d\n", sizeof(data));

	printf( "\ndata.i (before assigning data.f) : %d\n", data.i);

	data.f = 220.5;

	printf( "\ndata.i (after assigning data.f): %d\n", data.i);
	printf( "data.f (before assigning data.str): %f\n", data.f);

	/* Assigning the string "C Programming" to data.str */
	strcpy(data.str, "C Programming");

	printf( "\ndata.i (after assigning data.str): %d\n", data.i);
	printf( "data.f (after assigning data.str): %f\n", data.f);
	printf( "data.str : %s\n", data.str);

	return 0;
}
```

Output

```
Memory size occupied by data : 20

data.i (before assigning data.f) : 10

data.i (after assigning data.f): 1130135552
data.f (before assigning data.str): 220.500000

data.i (after assigning data.str): 1917853763
data.f (after assigning data.str): 4122360580327794900000000000000.000000
data.str : C Programming
```

Here, we can see that the values of **data.i** got corrupted after assigning values to **data.f**. Similarly both the values of **data.i** and **data.f** got corrupted after the assignment of string “C Programming” to **data.str**.

Difference between Structure and Union in C

Structure	Union
Keyword struct defines a structure.	Keyword union defines a union.
When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members.	When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to size of largest member.
Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Altering the value of member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
All member can be accessed at a time.	Only one member can be accessed at a time.

Qualifiers in C

Qualifiers in C alters the meaning of base data types to yield a new data type. Different types of qualifiers are as follows:

- **Size qualifiers:** Size qualifiers alter the size of a basic data type. There are two size qualifiers, `long` and `short`. For example:
  1. `long double x;` Here size of `double` is 8 bytes. However, when `long` keyword is used, that variable becomes 12 bytes.
  2. `short int x;` Here size of `int` is 4 bytes. However, when `short` keyword is used, that variable becomes 2 bytes.

  However we can’t use these qualifiers with all data types.

- **Sign qualifiers:** Integers and floating point variables can hold both negative and positive values. However, if a variable needs to hold positive value only, `unsigned` qualifier is used. For example: `unsigned int x;` There is another qualifier `signed` which can hold both negative and positive value. However, it is not necessary to define variable `signed` since a variable is signed by default. It is important to note that, sign qualifiers can be applied to `int` and `char` types only.

- **Constant qualifiers:** A variable can be declared as a constant. To do so `const` keyword is used. For example: `const int x = 20;` The value of `cost` cannot be changed in the program.

- **Volatile qualifiers:** A variable should be declared `volatile` whenever its value can be changed by some external sources outside the program.

- **Static qualifiers:** If we qualify a variable with the **`static`** keyword, the compiler will generate instructions so as to create the variable in the memory as soon as the execution of the process begins. Even before the execution control enters the entry point that is the main function. All **`static`** variables are also available / accessible from any location of the program throughout the lifecycle of the process.

Operators in C

An operator is a symbol which operates on a value or a variable. For example: + is an operator to perform addition. C programming has wide range of operators to perform various operations. For better understanding of operators, these operators can be classified as:

- **Arithmetic Operators:** An arithmetic operator performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).

Operator	**Meaning of Operator**
`+`	Addition or unary plus
`-`	Subtraction or unary minus
`*`	Multiplication
`/`	Division
`%`	Remainder after division

- **Increment and decrement operators:** C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1. ++ increases the value by 1 whereas -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example

```
#include <stdio.h>

int main()
{
      int a = 10, b = 20;
      float c = 3.6, d = 6.8;

      printf("++a = %d", ++a); // Pre-increment of a
      printf("\nb++ = %d", b++); // Post-increment of b
      // Value of b after Post-increment of b
      printf("\nb = %d", b);

      printf("\n++c = %g", ++c); // Pre-increment of c
      printf("\nd++ = %g", d++); // Post-increment of d
      // Value of d after Post-increment of d
      printf("\nd = %g", d);

      return 0;
}
```

Output

```
++a = 11
b++ = 20
b = 21
++c = 4.6
d++ = 6.8
d = 7.8
```

- **Assignment Operators:** An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Operator	Usage	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`

- **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 returns 0`
`>`	Greater than	`5 > 3 returns 1`
`<`	Less than	`5 < 3 returns 0`
`!=`	Not equal to	`5 != 3 returns 1`
`>=`	Greater than or equal to	`5 >= 3 returns 1`
`<=`	Less than or equal to	`5 <= 3 return 0`

- **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. Following table shows all the logical operators supported by C language.

Operator	Meaning of Operator	Example
`&&`	Logial 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.

- **Bitwise Operators:** During computation, mathematical operations like: addition, subtraction, addition and division 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

- **Other Operators**

Operator	Meaning of Operator	Example
`,`	Comma operators are used to link related expressions together.	**`int a, c = 5, d;`**
`sizeof`	The sizeof is an unary operator which returns the size of a variable or data type	**`sizeof(float)`** returns 4
`&`	Returns the address of an variable	**`&x`** returns address of the variable x
`*`	Pointer to a variable	**`*x`** will be pointer to a variable x
`? :`	It is actually the if condition that we use in C language decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.	If **`x = 10`** then **`x > 9 ? 1 : 0`** returns 1

Decision making in C

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements:

- **`if` statement:** The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are:
  1. **Simple `if` statement:** The general form of a simple **`if`** statement is

```
if(expression)
{
     statement-inside;
}
statement-outside;
```

If the expression returns true, then the statement-inside will be executed, otherwise statement-inside will be skipped and only the statement-outside will be executed.

2. **`if...else` statement:** The general form of a simple **`if...else`** statement is

   ```
   if(expression)
   {
         statement-block-1;
   }
   else
   {
         statement-block-2;
   }
   ```

   If the expression is true, the statement-block-1 will be executed, else statement-block-1 will be skipped and statement-block-2 will be executed.

3. **Nested `if....else` statement:** The general form of a nested **`if...else`** statement is

   ```
   if(expression)
   {
         if(expression1)
         {
               statement-block-1;
         }
         else
         {
               statement-block-2;
         }
   }
   else
   {
         statement-block-3;
   }
   ```

   If expression is false then statement-block-3 will be executed, otherwise the execution continues and enters inside the first **`if`** to perform the check for the next **`if`** block, where **`if`** expression-1 is true the statement-block-1 will be executed otherwise statement-block-2 will be executed.

4. **`else- if` ladder:** The general form of **`else-if`** ladder is

   ```
   if(expression1)
   {
         statement-block-1;
   }
   ```

```
else if(expression2)
{
      statement-block-2;
}
else if(expression3 )
{
      statement-block-3;
}
else
      default-statement;
```

The expressions will be tested from the top (of the ladder) to downwards. As soon as a true condition is found, the statement associated with it is executed.

- **`switch` statement:** **`switch`** statement is a control statement that allows us to choose only one choice among the many given choices. When we want to solve multiple option type problems, for example: Menu like program, where one value is associated with each option and we need to choose only one at a time, then, **`switch`** statement is used. The general form of **`switch`** statement is

```
switch(expression)
{
      case value-1:
            block-1;
            break;
      case value-2:
            block-2;
            break;
      case value-3:
            block-3;
            break;
      case value-4:
            block-4;
            break;
      default:
            default-block;
            break;
}
```

The expression in **`switch`** evaluates to return an integral value, which is then compared to the values present in different cases. It executes that block of code which matches the case value. If there is no match, then default block will be executed (if present).

- **`goto` statement:** A **`goto`** statement in C programming provides an unconditional jump from the **`goto`** to a labeled statement in the same function. The syntax for a **`goto`** statement in C is as follows:

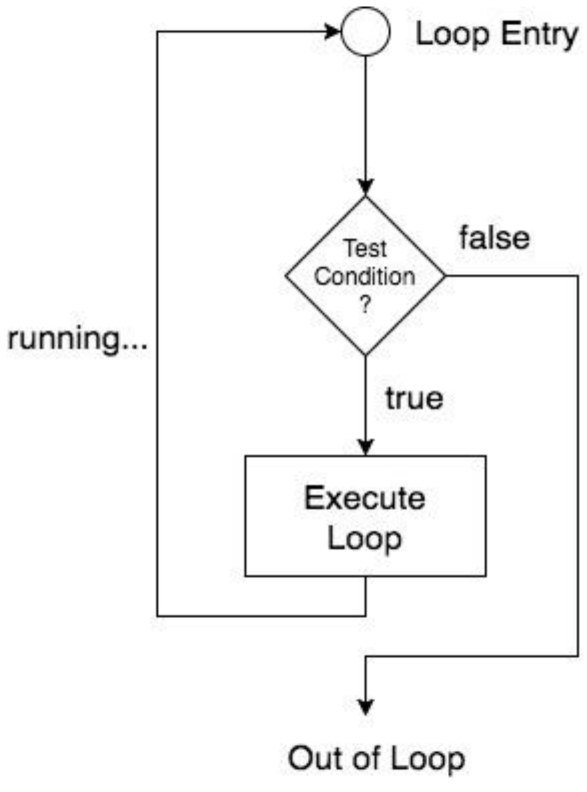
  ```
  goto label;
  ..
  .
  Labe_l: statement;
  ..
  .
  ```

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

  ***NOTE** - Use of **`goto`** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a **`goto`** can be rewritten to avoid them.*

Loops in C

We may encounter situations, when a block of code needs to be executed several number of times. In C language a loop statement allows us to execute a statement or a group of statements multiple times. Given below is the general form of a loop statement in C.

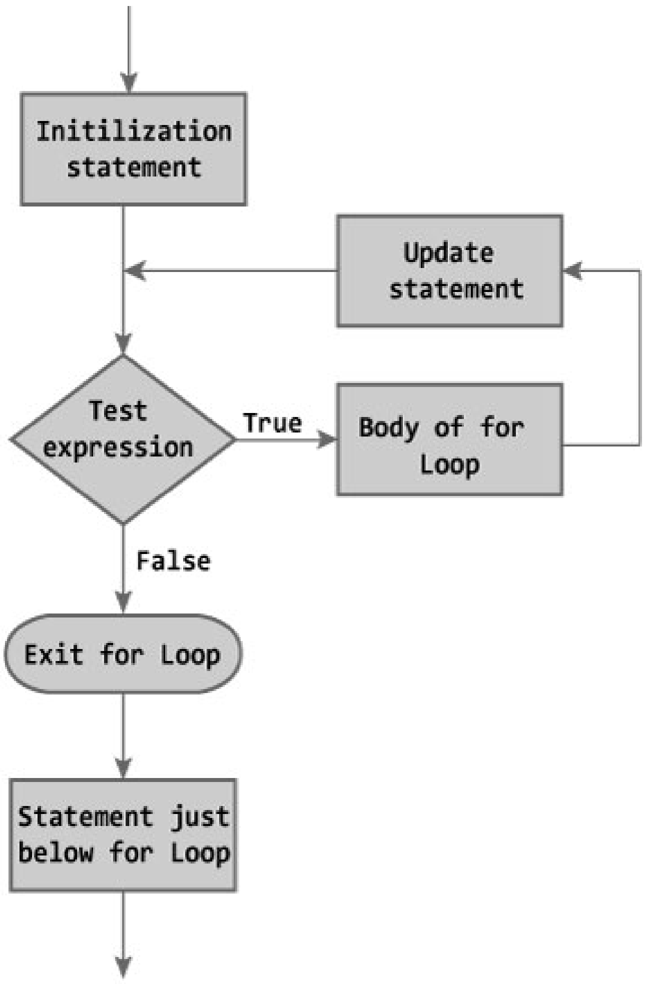


As per the above diagram, if the Test Condition is true, then the loop is executed, and if it is false then the execution breaks out of the loop. After the loop is successfully executed the execution again starts from the Loop entry and again checks for the Test condition, and this keeps on repeating.

Different types of Loops in C language are as follows:

- **`for` loop: `for`** loop executes a sequence of statements multiple times and abbreviates the code. The syntax of **`for`** loop is:

```
for(initializationStatement; testExpression; updateStatement)
{
     // codes
}
```

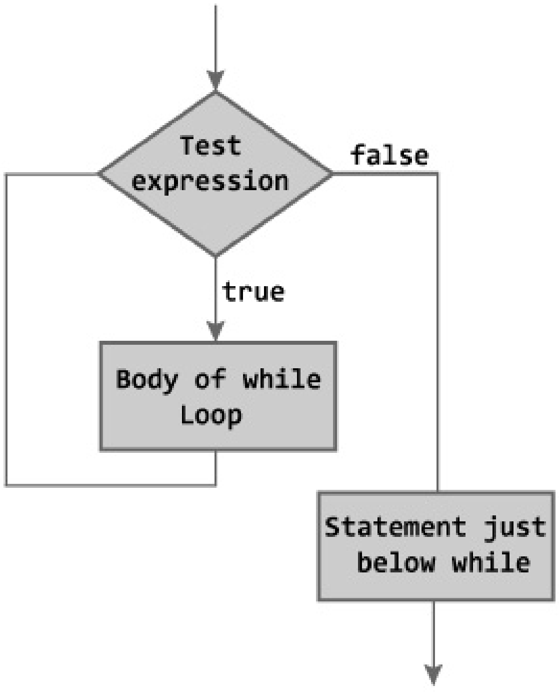


Here the initializationStatement is executed only once. Then, the testExpression is evaluated. If the test testExpression is false, **`for`** loop is terminated. But if the testExpression is true, codes inside the body of **`for`** loop is executed and the updateStatement is updated. This process repeats until the testExpression is false.

A **`for`** loop is commonly used when the number of iterations is known. However we can terminate a **`for`** loop with **`break`** statement for some terminating condition.

- **while loop: while** loop repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. The syntax of a **while** loop is:

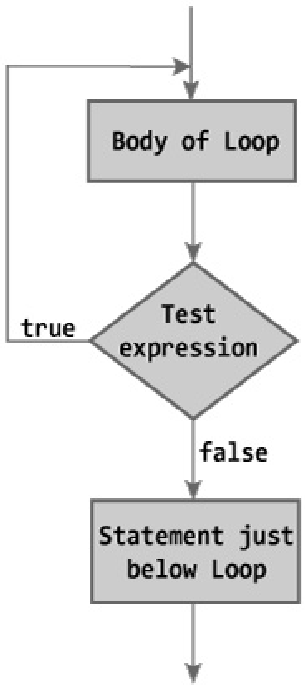
```
while(testExpression)
{
      //codes
}
```



Here the **while** loop evaluates the testExpression. If the testExpression is true, codes inside the body of **while** loop are exectued. The testExpression is evaluated again. The process goes on until the testExpression is false. When the testExpression is false, the **while** loop is terminated.

- **do..while loop: do..while** loop repeats a statement or group of statements while a given condition is true. It tests the condition at the end of the loop body. The syntax of a **do..while** loop is:

```
do
{
      // codes
}
while(testExpression);
```



Here the code block (loop body) inside the braces is executed once. Then, the testExpression is evaluated. If the test expression is true, the loop body is executed again. This process goes on until the evaluated testExpression is false. When the test expression is false, the **`do...while`** loop is terminated.

Functions in C

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 we can declare and define a group of statements once in the form of a function and it can be called and used whenever required. Every C program has at least one function, which is `main()`.

Benefits of Using Functions

1. It provides modularity to our program's structure.
2. It makes our code reusable. We 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 we use functions.
4. It makes the program more readable and easy to understand.

C functions can be classified into the following two categories:

- **Library functions:** Library functions are those functions which are already defined in C library. Example: `printf(), scanf(), log(), sin()` etc. We just need to include appropriate header files and libraries to use these functions.

Advantages of using C library functions

There are many library functions available in C programming for writing a good and efficient program. But, why should we use it? Below are the 4 most important advantages of using standard library functions.

1. One of the most important reasons you should use library functions is simply because they work. These functions have gone through multiple rigorous testing and are easy to use.
2. Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.
3. Since the general functions like printing to a screen, calculating the square root, and many more are already written. We shouldn't worry about creating them once again. It saves valuable time and our code may not always be the most efficient.
4. With ever changing real world needs, our application is expected to work every time, everywhere. These library functions help us in that they do the same thing on every computer. This saves time, effort and makes our program portable.

- **User-defined functions:** A User-defined functions 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. Below are the 4 most important advantages of using standard library functions.

Declaration of a User-defined function

Like any variable or an array, a function must also be declared before it’s used. Function declaration informs the compiler about the function name, parameters, and its return type. Functions should be declared in header files or before the definition of `main()` function. The actual body of the functions can be defined separately after `main()` function. General syntax for function declaration is,

```
return_type function_name(type_of_parameter1, type_of_parameter2, ......);
```

Here Function declaration consists of the following 4 parts:

1. **Return type:** When a function is declared to perform some sort of calculation or any operation, it is expected to provide us some result at the end. In such cases, a return statement is added at the end of function body. Return type specifies the type of value (**`int, float, char, double`**, etc.) that the function is expected to return to the program which called the function. In case our function doesn't return any value, the return type would be **`void`**.
2. **Function name:** Function name is an identifier and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

3. **Parameter list:** The parameter list declares the type and number of arguments that the function expects when it is called.
4. **Terminating semicolon:** It is used at the end of Function declaration.

Definition of a User-defined function

The general syntax of function definition is,

```
return_type function_name(data_type parameter1, data_type parameter2, ….. )
{
	// function body
}
```

The first line `return_type function_name(data_type parameter1, data_type parameter2, ….. )` is known as function header and the statement(s) within curly braces is called function body. It should be noted that while defining a function, there is no semicolon (;) after the parenthesis in the function header.

The function body contains the declarations and the statements (algorithm) necessary for performing the required task. The body is enclosed within curly braces `{ ... }` and consists of the following three parts.

1. Local variable declaration (if required).
2. Function statements to perform the task inside the function.
3. A return statement to return the result evaluated by the function (if return type is void, then no return statement is required).

Example

```
#include <stdio.h>

float max(float, float);
float min(float, float);

int main()
{
	float max_val, min_val;

	max_val = max(3, 7.8);
	min_val = min(3, 7.8);

	printf("Max. value is %g", max_val);
	printf("\nMin. value is %g", min_val);

	return 0;
}

float max(float x, float y)
{
	float result;

```

```
      if(x > y) result = x;
      else result = y;

      return result;
}

float min(float x, float y)
{
      float result;

      if(x < y) result = x;
      else result = y;

      return result;
}
```

Output

```
Max. value is 7.8
Min. value is 3
```

Calling a function

While creating a C function, we give a definition of what the function has to do. To use a function, we have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, we simply need to pass the required parameters along with the function name, and if the function returns a value, then we can store the returned value.

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function:

1. **Call by value:** The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the example of `swap()` function definition as follows:

```
#include <stdio.h>

void swap(int, int);

int main()
{
      int x = 10, y = 20;

      printf("Before swap, value of x : %d\n", x);
      printf("Before swap, value of y : %d\n", y);

      /* calling a function to swap the values */
      swap(x, y);

      printf("\nAfter swap, value of x : %d\n", x);
      printf("After swap, value of y : %d\n", y);

      return 0;
}

/* function definition to swap the values */
void swap(int a, int b)
{
      int temp;

      temp = a;
      a = b;
      b = temp;
}
```

Output

```
Before swap, value of x : 10
Before swap, value of y : 20

After swap, value of x : 10
After swap, value of y : 20
```

The output shows that there are no changes in the values of `x` & `y` after using of `swap()` function, though they had been changed inside the function.

2. **Call by reference:** The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the

address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly we need to declare the function parameters as pointer types as in the following example of swap() function, which exchanges the values of the two integer variables pointed to, by their arguments.

```
#include <stdio.h>

void swap(int *, int *);

int main()
{
     int x = 10, y = 20;

     printf("Before swap, value of x : %d\n", x);
     printf("Before swap, value of y : %d\n", y);

     /* calling a function to swap the values */
     swap(&x, &y);

     printf("\nAfter swap, value of x : %d\n", x);
     printf("After swap, value of y : %d\n", y);

     return 0;
}

/* function definition to swap the values */
void swap(int *a, int *b)
{
     int temp;

     temp = *a;
     *a = *b;
     *b = temp;
}
```

Output

```
Before swap, value of x : 10
Before swap, value of y : 20

After swap, value of x : 20
After swap, value of y : 10
```

The output shows that the values of `x` & `y` changes after calling of `swap()` function by reference.

Recursion in C functions

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows us to call a function inside the same function, then it is called a recursive call of the function.

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function; otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Examples of Recursive functions

1. Factorial of a given number.

```
#include <stdio.h>

unsigned long long int factorial(int);

int main()
{
	printf("5! = %llu\n", factorial(5));
	printf("10! = %llu\n", factorial(10));
	printf("20! = %llu\n", factorial(20));
	printf("40! = %llu\n", factorial(40));

	return 0;
}

/* Function definition for factorial */
unsigned long long int factorial(int i)
{
	if(i < 0) return 0;
	else if(i <= 1) return 1;
	else return i * factorial(i - 1);
}
```

Output

```
5! = 120
10! = 3628800
20! = 2432902008176640000
40! = 18376134811363311616
```

2. Sum of Natural Numbers.

```
#include <stdio.h>

unsigned long long int natural_sum(int);

int main()
{
      printf("1+2+...+10 = %llu\n", natural_sum(10));
      printf("1+2+...+100 = %llu\n", natural_sum(100));
      printf("1+2+...+1000 = %llu\n", natural_sum(1000));
      printf("1+2+...+10000 = %llu\n", natural_sum(10000));

      return 0;
}

/* Function definition for Sum of Natural Numbers */
unsigned long long int natural_sum(int i)
{
      if(i <= 0) return 0;
      else return i + natural_sum(i - 1);
}
```

Output

```
1+2+...+10 = 55
1+2+...+100 = 5050
1+2+...+1000 = 500500
1+2+...+10000 = 50005000
```

3. Fibonacci Series.

```
#include <stdio.h>

unsigned int fibonacci(int);

int main()
{
      int i;

      printf("The 10th term of Fibonacci Series is: %u\n\n",
                                                fibonacci(9));


      printf("The first 20 terms of Fibonacci Series are: ");
      for(i=0; i < 20; i++){
            printf("%u, ", fibonacci(i));
      }
```

```
	return 0;
}

/* Function definition for Fibonacci Series */
unsigned int fibonacci(int i)
{
	if(i <= 0) return 0;
	else if(i == 1) return 1;
	else return fibonacci(i-1) + fibonacci(i-2);
}
```

Output

```
The 10th term of Fibonacci Series is: 34

The first 20 terms of Fibonacci Series are: 0, 1, 1, 2, 3, 5,
8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
4181,
```

4. Counting total number of digits in an integer.

```
#include <stdio.h>

unsigned int count_digits(int);

int main()
{
	printf("Total digits in 12337: %d\n",
	                      count_digits(12337));
	printf("Total digits in 6: %d\n",
	                      count_digits(6));
	printf("Total digits in -980700: %d\n",
	                      count_digits(-980700));

	return 0;
}

/* Function definition to count digits */
unsigned int count_digits(int num)
{
	static int temp;

	temp = num/10;
	if(temp == 0) return 1;
	else return count_digits(temp) + 1;
}
```

Output

```
Total digits in 12337: 5
Total digits in 6: 1
Total digits in -980700: 6
```

5. Length of a string.

```
#include <stdio.h>

unsigned int string_length(char *);

int main()
{
      char *string1 = "MBB College";
      char *string2 = "Agartala";

      printf("Total number of characters in '%s' : %u\n",
                        string1, string_length(string1));
      printf("Total number of characters in '%s' : %u",
                        string2, string_length(string2));

      return 0;
}

/* Function definition to get string length */
unsigned int string_length(char *str)
{
      if(*str) return string_length(++str) + 1;
      else return 0;
}
```

Output

```
Total number of characters in 'MBB College' : 11
Total number of characters in 'Agartala' : 8
```

6. Lowest common multiple (LCM) of two numbers

```
#include <stdio.h>

int lcm(int, int);

int main()
```

```
{
	printf("LCM of 15 & 20: %d\n", lcm(15, 20));
	printf("LCM of 118 & 3: %d\n", lcm(118, 3));

	return 0;
}

/* Function definition to get LCM of two numbers */
int lcm(int a, int b)
{
	static int common = 1;

	if (common % a == 0 && common % b == 0) return common;
	common++;
	lcm(a, b);

	return common;
}
```

Output

```
LCM of 15 & 20: 60
LCM of 118 & 3: 354
```

7. Highest Common Factor (HCF) / Greatest Common Divisor (GCD) of two numbers

```
#include <stdio.h>

int hcf(int, int);

int main()
{
	printf("GCD/HCF of 15 & 20: %d\n", hcf(15, 20));
	printf("GCD/HCF of 118 & 3: %d\n", hcf(118, 3));

	return 0;
}

/* Function definition to get HCF/GCD of two numbers */
int hcf(int a, int b)
{
	if(a % b == 0) return b;
	else return hcf(b, a % b);
}
```

Output

```
GCD/HCF of 15 & 20: 5
GCD/HCF of 118 & 3: 1
```

8. Decimal to binary conversion

```
#include <stdio.h>

unsigned long long int int2binary(int);

int main()
{
      printf("Binary of 17: %llu\n", int2binary(17));
      printf("Binary of 1107: %llu\n", int2binary(1107));
      printf("Binary of 319: %llu\n", int2binary(319));

      return 0;
}

/* Function definition for Conversion of decimal to binary */
unsigned long long int int2binary(int num)
{
      if (num == 0) return 0;
      else return (num % 2) + 10 * int2binary(num / 2);
}
```

Output

```
Binary of 17: 10001
Binary of 1107: 10001010011
Binary of 319: 100111111
```

9. Binomial coefficient of two numbers (${}^{n}C_{x}$)

```
#include <stdio.h>

unsigned long long int ncx(int, int);

int main()
{
      printf("5C2: %llu\n", ncx(5, 2));
      printf("10C7: %llu\n", ncx(10, 7));
```

```
	return 0;
}

/* Function definition to Binomial coefficient of two numbers */
unsigned long long int ncx(int n, int x)
{
	if(x < 0 || x > n) return 0;
	else if(x == 0 || n == x) return 1;
	else return ncx(n-1, x) + ncx(n-1, x-1);
}
```

Output

```
5C2: 10
10C7: 120
```

10. Sorting by Quick-sort method.

```
#include <stdio.h>

void sort(float [], int, int);

int main()
{
	int i, n;
	float data[100];

	printf("Total observations: ");
	scanf("%d", &n);

	printf("\nInsert observations: ");
	for(i=0; i<n; i++) scanf("%f", &data[i]);

	sort(data, 0, n-1);

	printf("\nSorted observations:");
	for(i=0; i<n; i++) printf("\t%g", data[i]);

	return 0;
}

/* Function definition for Quick-sort method */
void sort(float x[], int start, int end)
{
	int i = start, j = end, k = (start+end)/2;
	float t, middle = x[k];

	do{
```

```
            while (x[i] < middle) i++;
            while (x[j] > middle) j--;
            if (i <= j) {
                 t = x[i];
                 x[i] = x[j];
                 x[j] = t;
                 i++;
                 j--;
            }
      }while (i <= j);
      if(start < j) sort(x,start,j);
      if(i < end) sort(x,i,end);
}
```

Output

```
Total observations: 10

Insert observations: 3 17 -5 8 7 10 1 8 12 2

Sorted observations: -5 1 2 3 7 8 8 10 12 17
```

Advantages and Disadvantages of Recursion

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is vital then, we should avoid using recursion. Recursions use more memory and are generally slow. Instead, we can use loop.

Variable Argument in C functions

Sometimes, we may come across a situation, when we want to have a function, which can take variable number of arguments / parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and we are allowed to define a function which can accept variable number of parameters based on our requirement. The following example shows the construction of such a function which can take the variable number of parameters and return their sum.

```

#include <stdio.h>
#include <stdarg.h>

double sum(int, ...);

int main() {
      printf("Sum of 2.3,3.5,4.6,5.0 = %g\n",
                                sum(4, 2.3,3.5,4.6,5.0));
      printf("Sum of 5.6,10.9,15.1 = %g\n",
                                sum(3, 5.6,10.9,15.1));

      return 0;
}

double sum(int num,...) {
      va_list x;
      int i;
      double sum = 0.0;

      /* initialize x for num number of arguments */
      va_start(x, num);

      /* access all the arguments assigned to valist */
      for(i = 0; i < num; i++) {
            sum += va_arg(x, double);
      }

      /* clean memory reserved for valist */
      va_end(x);

      return sum;
}
```

Output

```
Sum of 2.3,3.5,4.6,5.0 = 15.4
Sum of 5.6,10.9,15.1 = 31.6
```

It should be noted that the function `sum()` has been called twice and each time the first argument represents the total number of variable arguments being passed.

C - Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that region it cannot be accessed. There are three places where variables can be declared in C programming language:

- Inside a function or a block which is called local variables.
- Outside of all functions which is called global variables.
- In the definition of function parameters which are called formal parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of our program and they can be accessed inside any of the functions defined for the program. The following program show how global variables are used in a program.

```
#include <stdio.h>

/* Global variable declaration */
int x = 10, y = 200;

void change_global(int);

int main()
{
	printf("Value of x, y: %d, %d\n", x, y);
	x = 20;
	y = 400;
	printf("Value of x, y: %d, %d\n", x, y);
	change_global(30);
	printf("Value of x, y: %d, %d\n", x, y);

	return 0;
}

void change_global(int newx)
{
	/* Local variable declaration */
	int y;

	x = newx;
	y = newx;
}
```

Output

```
Value of x, y: 10, 200
Value of x, y: 20, 400
Value of x, y: 30, 400
```

Here we can see that a program can have same name (y) for local and global variables but the value of local variable inside a function will take preference.

Formal Parameters

Formal parameters are treated as local variables with-in a function and they take precedence over global variables.

Initialization of Local and Global Variables

When a local variable is defined, it is not initialized by the system; we must initialize it before use. Global variables are initialized automatically by the system when we define them as follows:

Data Type	Initial Default Value
`int`	0
`char`	'\0'
`float`	0
`double`	0
`pointer`	NULL

It is a good programming practice to initialize variables properly; otherwise our program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

Arrays in C

Array in C language is a collection or group of elements (data). All the elements of C array are homogeneous (similar). It has contiguous memory location.

C array is beneficial if we have to store similar elements. Suppose we have to store marks of 50 students, one way to do this is allotting 50 variables. So it will be typical and hard to manage. We can’t access the value of these variables with only 1 or 2 lines of code. In this case using array we can access these elements easily by defining a single variable. Only few lines of code are required to access the elements of array.

Arrays are of two types:

1. One-dimensional arrays
2. Multidimensional arrays

A one-dimensional array is like a list; A two dimensional array is like a table; The C language places no limits on the number of dimensions in an array. Some texts refer to one-dimensional arrays as vectors, two-dimensional arrays as matrices, and use the general term arrays when the number of dimensions is more than two.

Declaration of Array

To declare an array in C, we have to specify the type of the elements and the number of elements required by an array as follows:

```
data_type array_name[array_size];
```

This is called a single-dimensional array. The `array_size` must be an integer constant (not a C variable) greater than zero and `data_type` can be any valid C data type. For example:

```
int marks[100];
```

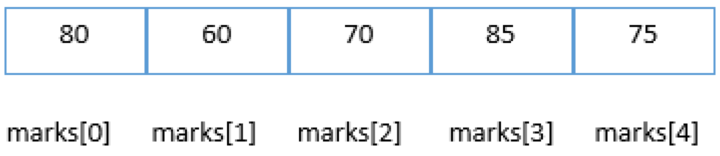
Here, **`int`** is the `data_type`, `marks` is the `array_name` and 100 is the `array_size`.

A multi-dimensional array can be declared as follows:

```
data_type array_name[size_of_dim_1][size_of_dim_2]…..[ size_of_dim_n];
```

Elements of an Array

We can access elements of an array by indices. Suppose we declared an array `marks` as above. The first element is `marks[0]`, second element is `marks[1]` and so on.



It should be noted that

1. Arrays have 0 as the first index not 1.
2. If the size of an array is *n*, the index of last element is *n-1* not *n*.
3. If the size of an array is 10, we can use the array elements from 0 to 9. If we try to access array elements outside of its bound, i.e. index greater than 9, the compiler may not show any error. However, this may cause unexpected output (undefined behavior).

Example

Here will obtain the Mean Deviation about Mean of a data-set using array.

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

int main()
{
      int i, n;

      /*The one dimensional data array 'x' with a capacity 100 elements*/
      float x[100];

      float mean = 0, MD = 0;

      printf("Total observations: ");
      scanf("%d", &n);

      printf("\nObservations: ");
      for(i=0; i<n; i++){
            scanf("%f", &x[i]);
            mean += x[i];
      }

      mean = mean / n;

      for(i=0; i<n; i++){
            MD += fabs(x[i] - mean);
      }

      MD = MD / n;

      printf("\nMean Deviation about Mean = %g", MD);

      return 0;
}
```

Output

```

Total observations: 5

Observations: 12 23 7 45 10

Mean Deviation about Mean = 11.68
```

Advantage of Array

1. Code Optimization: Less code to the access the data.
2. Easy to traverse data: By using the for loop, we can retrieve the elements of an array easily.
3. Easy to sort data: To sort the elements of array, we need a few lines of code only.
4. Random Access: We can access any element randomly using the array.

Disadvantage of Array

Fixed Size: Whatever size, we define at the time of declaration of array, we can't exceed the limit. So, it doesn't grow the size dynamically.

Array as function parameter

Whenever we need to pass a list of elements as argument to any function in C language, it is preferred to do so using an array. If we want to pass a single-dimension array as an argument in a function, we should have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results. Similarly, we can pass multi-dimensional arrays as formal parameters.

1. Formal parameters as an sized array

```
        void my_function (int param[10]) {
          .
          .
      }
```

2. Formal parameters as an un-sized array

```
      void my_function (int param[]) {
          .
          .
      }
```

3. Formal parameters as a pointer

```
      void my_function (int *param) {
          .
          .
      }
```

Now, consider the following example, where the `average(..)` function takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array.

```
#include <stdio.h>

float average(float [], int);

int main()
{
      /*The one dimensional data array 5 elements*/
      float data[5] = {10, 23, 37, 41, 50};

      printf("\nAverage = %g", average(data, 5));

      return 0;
}

/* Function to obtain average of some values */
float average(float x[], int n)
{
      int i;
      float avg = 0;

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

      avg = avg/n;

      return avg;
}
```

Output

```
Average = 32.2
```

Return array from function

C programming does not allow returning an entire array from a function. However, you can return a pointer to an array by specifying the array's name without an index. While returning an address of a local variable from a function, it should be remember that we have to define the local variable as static variable.

Consider the following example where a function returns a pointer to an array of random numbers.

```
#include <stdio.h>

int *rand_numbers();

main()
{
	int i, *rnd;

	rnd = rand_numbers();

	printf("\n10 random numbers: ");
	for(i=0; i<10; i++) printf(" %d", rnd[i]);

	return 0;
}

/* Function to generate and return random numbers */
int *rand_numbers()
{
	int i;
	static int r[10];

	/* set the seed of random numbers*/
	srand((unsigned int)time(NULL));

	for(i=0; i<10; i++) r[i] = rand();

	return r;
}
```

Output

```
10 random numbers:  9644 18684 1554 1009 11360 18594 32 3593 31637 4038
```

Pointers

A Pointer in C language is a variable which holds the address of another variable of same data type. Pointers are used in C program to access the memory and manipulate the address. Pointers are one of the most distinct and exciting features of C and C++ programming language that differentiates it from other popular programming languages like: Java and Python. Before we start understanding what pointers are and what they can do, we have to understand “Address of a memory location”.

Address in C

Whenever a variable is defined in C language, a memory location is assigned for it, in which its value will be stored. We can easily check this memory address, using the **`&`** operator. If **`x`** is the name of the variable, then **`&x`** will give it's address. Here **`&`** is known as reference operator. In the **`scanf()`** function we use it to store the user inputted value of a variable in the address of that variable like **`scanf(“%d”, &x);`**

Example

```
#include <stdio.h>

int main()
{
      int x = 5;

      printf("Value: %d\n", x);
      printf("Address: %u", &x);

      return 0;
}
```

Output

```
Value: 5
Address: 2293572
```

In above source code, value 5 is stored in the memory location 2293572. x is just the name given to that location. We may obtain different value of address while using this code.