C programming syllabus

Chapter 1: C fundamentals

* History of c
* C character set
* Identifiers and keywords
* Data types
* Variables
* Declarations
* Constants
* Expressions
* Instructions
* The first c program
* Compilation and Execution

Chapter 2: Operators and data input/output functions

* Operators
  + Arithmetic
  + Assignment
  + Unary
  + Relational and Logical
  + Conditional and comm.
* Input and Output Library Functions
  + Printf()
  + Scanf()
  + getchar()
  + putchar()
  + gets()
  + puts()

Chapter 3: Control Statement

* Decision Making and Branching
  + If Statement(if, if-else, if else ladder, nested if else)
  + Switch, continue ,break, goto statement
* Decision making looping
  + While ,do-while, for statement
  + Nested loops

Chapter 4: Functions

* Defining a function, Accessing a function
* Passing Argument to the function(call by value ,call by reference) specifying argument data types
* Scope and lifetime of variables
* Function prototypes
* Category of function(types 4)
* Recursion

Chapter 5: Array

* Defining an array
* One Dimensional array, Declaration and initialization of Array
* Two Dimensional array Declaration and initialization of array
* Passing arrays to a function

Chapter 6: Characters and strings

* The char data type, using character data type, using string
* Declaring and initializing string variable
* Reading String from terminal
* Writing Strings to screen, putting Strings together

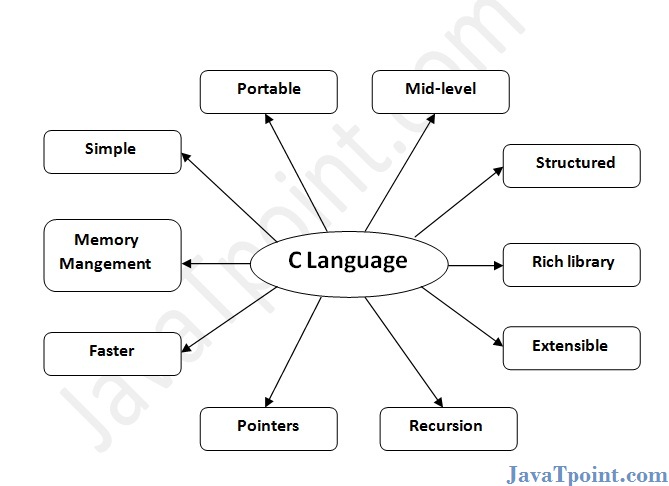
Chapter 1: C Fundamentals

* History of c
  + **C programming language** was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.
  + **Dennis Ritchie** is known as the **founder of the c language**.
  + It was developed to overcome the problems of previous languages such as B, BCPL, etc.
  + Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.

|  |  |  |
| --- | --- | --- |
| **Language** | **Year** | **Developed By** |
| Algol | 1960 | International Group |
| BCPL | 1967 | Martin Richard |
| B | 1970 | Ken Thompson |
| Traditional C | 1972 | Dennis Ritchie |
| K & R C | 1978 | Kernighan & Dennis Ritchie |
| ANSI C | 1989 | ANSI Committee |
| ANSI/ISO C | 1990 | ISO Committee |
| C99 | 1999 | Standardization Committee |

**next>><<prev**

Features of C Language



C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

# C Character Set

As every language contains a set of characters used to construct words, statements etc., C language also has a set of characters which include **alphabets, digits** and **special symbols**. C language supports a total of 256 characters.  
  
Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set. C language character set contains the following set of characters...

1. Alphabets
2. Digits
3. Special Symbols

**Alphabets**

C language supports all the alphabets from english language. Lower and upper case letters together supports 52 alphabets.

lower case letters - **a to z**

UPPER CASE LETTERS - **A to Z**

**Digits**

C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

**Special Symbols**

C language supports rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, back spaces and other special symbols.

Special Symbols - **~ @ # $ % ^ & \* ( ) \_ - + = { } [ ] ; : ' " / ? . > , < \ | tab newline space NULL bell backspace verticaltab etc.,**

Every character in C language has its equivalent ASCII (American Standard Code for Information Interchange) value.

**C Identifiers**

In C programming language, programmers can specify their own name to variable, array, pointer, function, lable etc... Identifier is a collection of characters which acts as name of variable, function, array, pointer, strcture, lable etc... In other words, an identifier can be defined as user defined name to identify an enity uniquely in c programming language that name may be of variable name, function name, array name, pointer name, structure name or a lable.

**Identifier is a user defined name of an entity to identify it uniquely during the program execution**

**Example**

**Int marks;  
char studentName[30];**  
Here, **marks** and **studentName** are identifiers.

**Rules for Creating Identifiers**

1. An identifier can contain **letters** (UPPERCASE and lowercase), **numerics** & **underscore** symbol only.
2. An identifier should not start with numerical value. It can start with a letter or an underscore.
3. We should not use any special symbols in between the identifier even whitespace. However, the only underscore symbol is allowed.
4. Keywords should not be used as identifiers.
5. There is no limit for length of an identifier. However, compiler consider first 31 characters only.
6. An identifier must be unique in its scope.

# C Keywords

In C programming language, keywords are special words with predefined meaning. Keywords are also known as resevered words in C programming language.  
In C programming language, there are **32 keywords**. All the 32 keywords has their own meaning which is already known to the compiler.

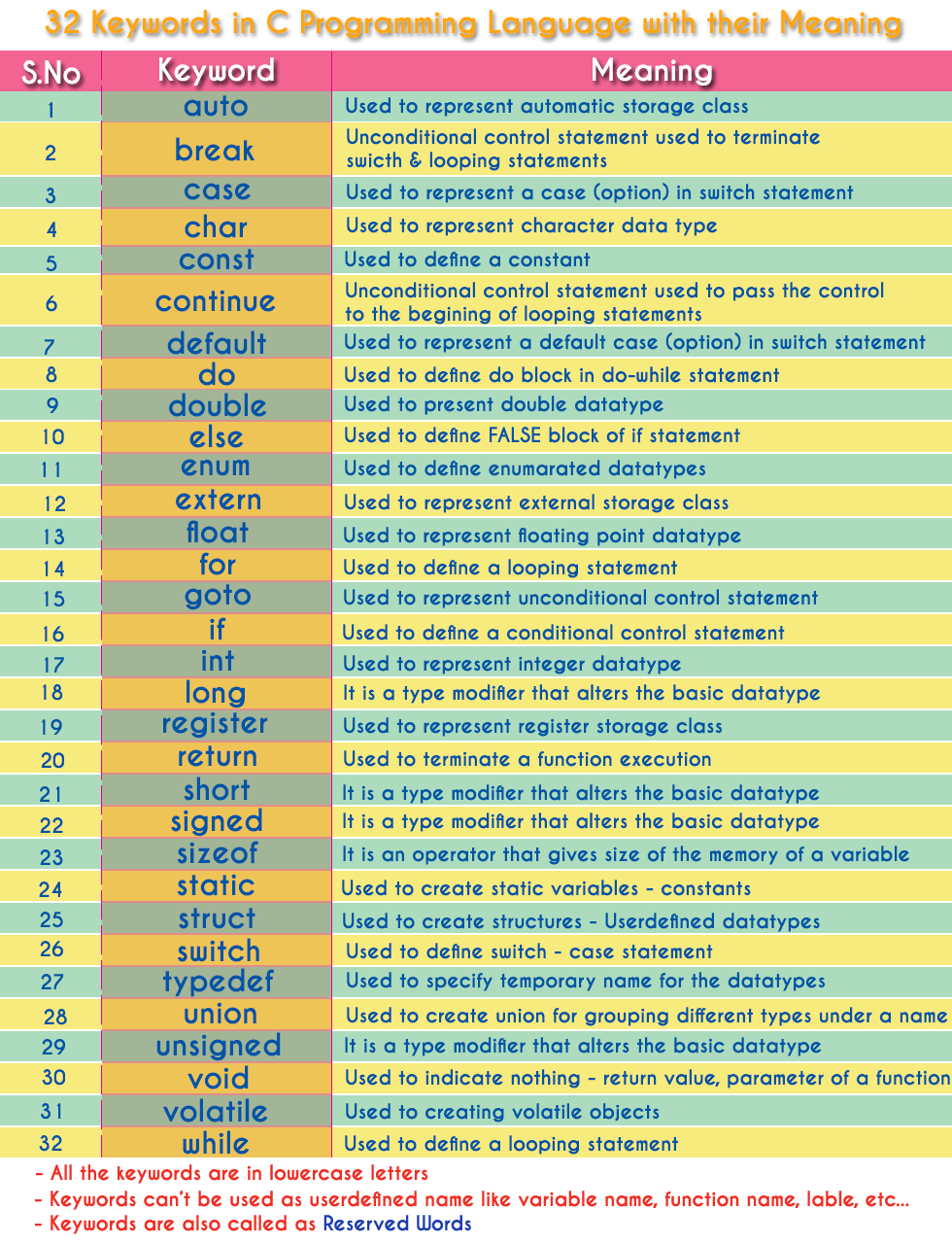
**Keywords are the reserved words with predefined meaning which already known to the compiler**

Whenever C compiler come across a keyword, automatically it understands its meaning.

**Properties of Keywords**

1. All the keywords in C programming language are defined as lowercase letters so they must be use only in lowercase letters
2. Every keyword has a specific meaning, users can not change that meaning.
3. Keywords can not be used as user defined names like variable, functions, arrays, pointers etc...
4. Every keyword in C programming language, represents something or specifies some kind of action to be performed by the compiler.

The following table specifies all the 32 keywords with their meaning...



## Data types in C Language

Data types specify how we enter data into our programs and what type of data we enter. C language supports 2 different type of data types:

1. **Primary data types**:

These are fundamental data types in C namely integer(int), floating point(float), character(char) and void.

1. **Derived data types**:

Derived data types are nothing but primary datatypes but a little twisted or grouped together like **array**, **stucture**, **union** and **pointer**.

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.

### Integer type

Integers are used to store whole numbers.

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

|  |  |  |
| --- | --- | --- |
| **Type** | **Size(bytes)** | **Range** |
| int or signed int | 2 | -32,768 to 32767 |
| unsigned int | 2 | 0 to 65535 |
| short int or signed short int | 1 | -128 to 127 |
| unsigned short int | 1 | 0 to 255 |
| long int or signed long int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 | 0 to 4,294,967,295 |

### Floating point type

Floating types are used to store real numbers.

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

|  |  |  |
| --- | --- | --- |
| **Type** | **Size(bytes)** | **Range** |
| Float | 4 | 3.4E-38 to 3.4E+38 |
| Double | 8 | 1.7E-308 to 1.7E+308 |
| long double | 10 | 3.4E-4932 to 1.1E+4932 |

### Character type

Character types are used to store characters value.

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

|  |  |  |
| --- | --- | --- |
| **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.

## Variables in C Language

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.

* 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.
* void

### 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.

**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;

**Constants**

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

const data\_type variable\_name;

(or) const data\_type \*variable\_name;

**TYPES OF C CONSTANT:**

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

**4. BACKSLASH CHARACTER CONSTANTS IN C:**

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

|  |  |
| --- | --- |
| **Backslash\_character** | **Meaning** |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Horizontal tab |
| \” | Double quote |
| \’ | Single quote |
| \\ | Backslash |
| \v | Vertical tab |
| \a | Alert or bell |
| \? | Question mark |
|  |  |

**HOW TO USE CONSTANTS IN A C PROGRAM?**

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

1. By “const” keyword
2. By “#define” preprocessor directive

## C Programming Expression

In programming, an expression is any legal combination of symbols that represents a value.

C Programming code gets compiled firstly before execution. In the different phases of compiler, c programming expression is checked for its validity.

| **Expressions** | **Validity** |
| --- | --- |
| a + b | Expression is valid since it contain + operator which is binary operator |
| + + a + b | Invalid Expression |

## Types of Expression :

In Programming, different verities of expressions are given to the compiler. Expressions can be classified on the basis of Position of Operators in an expression –

| **Type** | **Explanation** | **Example** |
| --- | --- | --- |
| Infix | Expression in which Operator is in between Operands | a + b |
| Prefix | Expression in which Operator is written before Operands | + a b |
| Postfix | Expression in which Operator is written after Operands | a b + |

### C Input and Output

**Input** means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

### scanf() and printf() functions

The standard input-output header file, named stdio.h contains the definition of the functions printf() and scanf(), which are used to display output on screen and to take input from user respectively.

|  |  |
| --- | --- |
| **Format String** | **Meaning** |
| %d | Scan or print an integer as signed decimal number |
| %f | Scan or print a floating point number |
| %c | To scan or print a character |
| %s | To scan or print a character string. The scanning ends at whitespace. |

C Instructions

There are basically three types of instructions in C:

* **Type Declaration Instruction** – To declare the type of variables used in a C program.
* **Arithmetic Instruction** – To perform arithmetic operations between constants and variables
* **Control Instruction** – To control the sequence of execution of various statements in a C program.

## ****Type Declaration Instruction****

* This instruction is used to declare the type of variables being used in the program.
* Any variable used in the program must be declared before using it in any statement.
* The type declaration statement is written at the beginning of main( ) function.

int a;

int b;

**Arithmetic Instruction**

A C arithmetic instruction consists of a variable name on the left hand side of = and variable names & constants on the right hand side of =. The variables and constants appearing on the right hand side of = are connected by arithmetic operators like +, -, \*, and /.

C=a+b;

compilation and execution

The compilation and execution process of C can be divided in to multiple steps:

* Preprocessing - Using a Preprocessor program to convert C source code in expanded source code. "#includes" and "#defines" statements will be processed and replaced actually source codes in this step.
* Compilation - Using a Compiler program to convert C expanded source to assembly source code.
* Assembly - Using a Assembler program to convert assembly source code to object code.
* Linking - Using a Linker program to convert object code to executable code. Multiple units of object codes are linked to together in this step.
* Loading - Using a Loader program to load the executable code into CPU for execution.

Chapter 2

## Operators in C Language

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation. Operators are used in programs to manipulate data and variables.

C operators can be classified into following types:

* Arithmetic operators
* Relational operators
* Logical operators
* Bitwise operators
* Assignment operators
* Conditional operators
* Special operators

### Arithmetic operators

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + | adds two operands |
| - | subtract second operands from first |
| \* | multiply two operand |
| / | divide numerator by denominator |
| % | remainder of division |
| ++ | Increment operator - increases integer value by one |
| -- | Decrement operator - decreases integer value by one |

### Relational operators

The following table shows all relation operators supported by C.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | Check if two operand are equal |
| != | Check if two operand are not equal. |
| > | Check if operand on the left is greater than operand on the right |
| < | Check operand on the left is smaller than right operand |
| >= | check left operand is greater than or equal to right operand |
| <= | Check if operand on left is smaller than or equal to right operand |

### Logical operators

C language supports following 3 logical operators. Suppose a = 1 and b = 0,

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Logical AND | (a && b) is false |
| || | Logical OR | (a || b) is true |
| ! | Logical NOT | (!a) is false |

### Bitwise operators

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to float or double(These are datatypes, we will learn about them in the next tutorial).

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |
| << | left shift |
| >> | right shift |

Now lets see truth table for bitwise &, | and ^

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a** | **b** | **a & b** | **a | b** | **a ^ b** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

The bitwise **shift** operator, shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value have to be shifted. Both operands have the same precedence.

**Example** :

a = 0001000

b = 2

a << b = 0100000

a >> b = 0000010

### Assignment Operators

Assignment operators supported by C language are as follows.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | assigns values from right side operands to left side operand | a=b |
| += | adds right operand to the left operand and assign the result to left | a+=b is same as a=a+b |
| -= | subtracts right operand from the left operand and assign the result to left operand | a-=b is same as a=a-b |
| \*= | mutiply left operand with the right operand and assign the result to left operand | a\*=b is same as a=a\*b |
| /= | divides left operand with the right operand and assign the result to left operand | a/=b is same as a=a/b |
| %= | calculate modulus using two operands and assign the result to left operand | a%=b is same as a=a%b |

### Conditional operator

The conditional operators in C language are known by two more names

1. **Ternary Operator**
2. **? : Operator**

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.

The syntax of a conditional operator is :

expression 1 ? expression 2: expression 3

**Explanation:**

**A= b>c?b:c;**

* The question mark **"?"** in the syntax represents the **if** part.
* The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
* If (expression 1) returns true then the expression on the left side of **" : "** i.e (expression 2) is executed.
* If (expression 1) returns false then the expression on the right side of **" : "** i.e (expression 3) is executed.

### Special operator

|  |  |  |
| --- | --- | --- |
| **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** |

### C Input and Output

**Input** means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

In this tutorial, we will learn about such functions, which can be used in our program to take input from user and to output the result on screen.

All these built-in functions are present in C header files, we will also specify the name of header files in which a particular function is defined while discussing about it.

### scanf() and printf() functions

The standard input-output header file, named stdio.h contains the definition of the functions printf() and scanf(), which are used to display output on screen and to take input from user respectively.

#include<stdio.h>

void main()

{

// defining a variable

int i;

/\*

displaying message on the screen

asking the user to input a value

\*/

printf("Please enter a value...");

/\*

reading the value entered by the user

\*/

scanf("%d", &i);

/\*

displaying the number as output

\*/

printf( "\nYou entered: %d", i);

}

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered on screen.

You must be wondering what is the purpose of %d inside the scanf() or printf() functions. It is known as **format string** and this informs the scanf() function, what type of input to expect and in printf() it is used to give a heads up to the compiler, what type of output to expect.

|  |  |
| --- | --- |
| **Format String** | **Meaning** |
| %d | Scan or print an integer as signed decimal number |
| %f | Scan or print a floating point number |
| %c | To scan or print a character |
| %s | To scan or print a character string. The scanning ends at whitespace. |

We can also **limit the number of digits or characters** that can be input or output, by adding a number with the format string specifier, like "%1d" or "%3s", the first one means a single numeric digit and the second one means 3 characters, hence if you try to input 42, while scanf() has "%1d", it will take only 4 as input. Same is the case for output.

In C Language, computer monitor, printer etc output devices are treated as files and the same process is followed to write output to these devices as would have been followed to write the output to a file.

**NOTE :** printf() function returns the number of characters printed by it, and scanf() returns the number of characters read by it.

int i = printf("studytonight");

In this program printf("studytonight"); will return 12 as result, which will be stored in the variable i, because studytonight has 12 characters.

### getchar() & putchar() functions

The getchar() function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in a [loop](https://www.studytonight.com/c/loops-in-c.php) in case you want to read more than one character. The putchar() function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use putchar() method in a loop.

#include <stdio.h>

void main( )

{

int c;

printf("Enter a character");

/\*

Take a character as input and

store it in variable c

\*/

c = getchar();

/\*

display the character stored

in variable c

\*/

putchar(c);

}

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

### gets() & puts() functions

The gets() function reads a line from **stdin**(standard input) into the buffer pointed to by str [pointer](https://www.studytonight.com/pointers-in-c.php), until either a terminating newline or EOF (end of file) occurs. The puts() function writes the string str and a trailing newline to **stdout**.

str → This is the pointer to an array of chars where the C string is stored. (Ignore if you are not able to understand this now.)

#include<stdio.h>

void main()

{

/\* character array of length 100 \*/

char str[100];

printf("Enter a string");

gets( str );

puts( str );

getch();

}

When you will compile the above code, it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

### Difference between scanf() and gets()

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.

If you enter name as **Study Tonight** using scanf() it will only read and store **Study** and will leave the part after space. But gets() function will read it completely.

Chapter 3

## 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
* switch statement
* conditional operator statement (? : operator)
* goto statement

### Decision making with 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
2. if....else statement
3. Nested if....else statement
4. Using else if statement

#### 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** is skipped and only the **statement-outside** is executed.

**Example:**

#include <stdio.h>

void main( )

{

int x, y;

x = 15;

y = 13;

if (x > y )

{

printf("x is greater than y");

}

}

x is greater than y

#### if...else statement

The general form of a simple if...else statement is,

if(expression)

{

statement block1;

}

else

{

statement block2;

}

If the *expression* is true, the **statement-block1** is executed, else **statement-block1** is skipped and **statement-block2** is executed.

**Example:**

#include <stdio.h>

void main( )

{

int x, y;

x = 15;

y = 18;

if (x > y )

{

printf("x is greater than y");

}

else

{

printf("y is greater than x");

}

}

y is greater than x

#### Nested if....else statement

The general form of a nested if...else statement is,

if( expression )

{

if( expression1 )

{

statement block1;

}

else

{

statement block2;

}

}

else

{

statement block3;

}

if *expression* is false then **statement-block3** 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-block1** is executed otherwise **statement-block2** is executed.

**Example:**

#include <stdio.h>

void main( )

{

int a, b, c;

printf("Enter 3 numbers...");

scanf("%d%d%d",&a, &b, &c);

if(a > b)

{

if(a > c)

{

printf("a is the greatest");

}

else

{

printf("c is the greatest");

}

}

else

{

if(b > c)

{

printf("b is the greatest");

}

else

{

printf("c is the greatest");

}

}

}

#### else if ladder

The general form of else-if ladder is,

if(expression1)

{

statement block1;

}

else if(expression2)

{

statement block2;

}

else if(expression3 )

{

statement block3;

}

else

default statement;

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

**Example :**

#include <stdio.h>

void main( )

{

int a;

printf("Enter a number...");

scanf("%d", &a);

if(a%5 == 0 && a%8 == 0)

{

printf("Divisible by both 5 and 8");

}

else if(a%8 == 0)

{

printf("Divisible by 8");

}

else if(a%5 == 0)

{

printf("Divisible by 5");

}

else

{

printf("Divisible by none");

}

}

### Points to Remember

1. In if statement, a single statement can be included without enclosing it into curly braces { ... }
2. int a = 5;
3. if(a > 4)

printf("success");

No curly braces are required in the above case, but if we have more than one statement inside ifcondition, then we must enclose them inside curly braces.

1. == must be used for comparison in the expression of if condition, if you use = the expression will always return **true**, because it performs assignment not comparison.
2. Other than **0(zero)**, all other values are considered as **true**.
3. if(27)

printf("hello");

In above example, **hello** will be printed.

## Switch statement in C

When you want to solve multiple option type problems, for example: Menu like program, where one value is associated with each option and you need to choose only one at a time, then, switchstatement is used.

Switch statement is a control statement that allows us to choose only one choice among the many given choices. 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 is executed(if present). 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;

}

### Rules for using switch statement

1. The expression (after switch keyword) must yield an **integer** value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
2. The case **label** values must be unique.
3. The case label must end with a colon(:)
4. The next line, after the **case** statement, can be any valid C statement.

### Points to Remember

1. We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.
2. break statements are used to **exit** the switch block. It isn't necessary to use break after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block.
3. int i = 1;
4. switch(i)
5. {
6. case 1:
7. printf("A"); // No break
8. case 2:
9. printf("B"); // No break
10. case 3:
11. printf("C");
12. break;

}

A B C

The output was supposed to be only **A** because only the first case matches, but as there is no break statement after that block, the next blocks are executed too, until it a break statement in encountered or the execution reaches the end of the switch block.

1. **default** case is executed when none of the mentioned case matches the switch expression. The default case can be placed anywhere in the switch case. Even if we don't include the default case, switch statement works.
2. Nesting of switch statements are allowed, which means you can have switch statements inside another switch block. However, nested switch statements should be avoided as it makes the program more complex and less readable.

### Example of switch statement

#include<stdio.h>

void main( )

{

int a, b, c, choice;

while(choice != 3)

{

/\* Printing the available options \*/

printf("\n 1. Press 1 for addition");

printf("\n 2. Press 2 for subtraction");

printf("\n Enter your choice");

/\* Taking users input \*/

scanf("%d", &choice);

switch(choice)

{

case 1:

printf("Enter 2 numbers");

scanf("%d%d", &a, &b);

c = a + b;

printf("%d", c);

break;

case 2:

printf("Enter 2 numbers");

scanf("%d%d", &a, &b);

c = a - b;

printf("%d", c);

break;

default:

printf("you have passed a wrong key");

printf("\n press any key to continue");

}

}

}

### Difference between switch and if

* if statements can evaluate float conditions. switch statements cannot evaluate floatconditions.
* if statement can evaluate relational operators. switch statement cannot evaluate relational operators i.e they are not allowed in switch statement.

# break, continue and goto statements

The **break;** **continue;** and **goto;** statements are used to alter the normal flow of a program.  
Loops perform a set of repetitive task until text expression becomes false but it is sometimes desirable to skip some statement/s inside loop or terminate the loop immediately without checking the test expression. In such cases, break and continue statements are used.

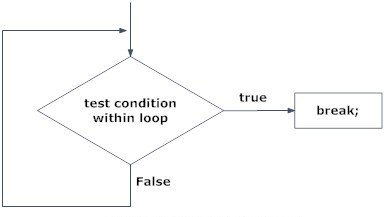
## break statement

In C programming, break statement is used with conditional if statement.  
The break is used in terminating the loop immediately after it is encountered.  
it is also used in switch...case statement. which is explained in next topic.

### *Syntax:*

**break;**

The break statement can be used in terminating loops like **for**, **while** and **do...while**



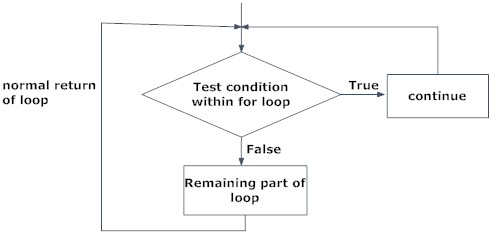
## continue statement

It is sometimes desirable to skip some statements inside the loop. In such cases, continue statement is used.

### *Syntax:*

**continue;**

Just like break, continue is also used with conditional if statement.



### *Example:*

**//Write a C Program Which use of continue statment.**

**#include<stdio.h>**

**#include<conio.h>**

**void main(){**

**int i, n=20;**

**clrscr();**

**for(i=1;i<=n;++i){**

**if(i % 5 == 0) {**

**printf("pass\n");**

**continue;      /\*this continue the execution of loop if i % 5 == 0 \*/**

**}**

**printf("%d\n",i);**

**}**

**getch();**

**}**

## goto statement

In C programming, goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

### *Syntax:*

**goto label;**

**.............**

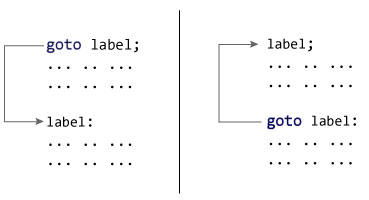
**.............**

**.............**

**label:**

**statement;**

In this syntax, **label** is an identifier.  
When, the control of program reaches to goto statement, the control of the program will jump to the **label:** and executes the code below it.



### *Example:*

**//Write a C Program Which Print 1 To 10 Number Using goto statement.**

**#include<stdio.h>**

**#include<conio.h>**

**void main()**

**{**

**int i=1;**

**clrscr();**

**count:              //This is Label**

**printf("%d\n",i);**

**i++;**

**if(i<=10) {**

**goto count;     //This jumps to label "count:"**

**}**

**getch();**

**}**

## ****Loops in C****

Loop is used to execute the block of code several times according to the condition given in the loop. It means it execute same code multiple times so it saves code and also helps to traverse the elements of array.

There are 3 types of loop –

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

**1. while Loop –**

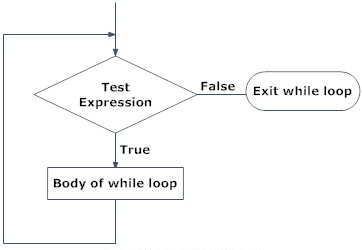
While loop execute the code until condition is false.

while(condition)

{

//code

}

[](https://cdn.intellipaat.com/wp-content/uploads/2015/11/while-loop1.png)

**2. do – while loop**

It also executes the code until condition is false. In this at least once, code is executed whether condition is true or false but this is not the case with while. While loop is executed only when the condition is true.

**Syntax**

do

{

//code

}while(condition);

**3. for Loop**

It also executes the code until condition is false. In this three parameters are given that is

* Initialization
* Condition
* Increment/Decrement

**Syntax**

for(initialization;condition;increment/decrement)

{

//code

}

It is used when number of iterations are known where while is used when number of iterations are not known.

**e.g.**

#include<stdio.h>

#include<conio.h>

void main()

{

int i;

for( i = 20; i < 25; i++) {

printf ("%d " , i);

}

getch();

}

# Nested loop in C

A loop inside another loop is called a nested loop. The depth of nested loop depends on the complexity of a problem. We can have any number of nested loops as required. Consider a nested loop where the outer loop runs n times and consists of another loop inside it. The inner loop runs m times. Then, the total number of times the inner loop runs during the program execution is n\*m.

## Types of nested loops

* [Nested while loop](https://www.programtopia.net/c-programming/docs/nested-loop#while)
* [Nested do-while loop](https://www.programtopia.net/c-programming/docs/nested-loop#do-while)
* [Nested for loop](https://www.programtopia.net/c-programming/docs/nested-loop#for)

**Note**: There can be mixed type of nested loop i.e. a for loop inside a while loop, or a while loop inside a do-while loop.

## Nested while loop

A while loop inside another while loop is called nested while loop.

### Syntax of Nested while loop

while (condition1)

{

  statement(s);

  while (condition2)

{

  statement(s);

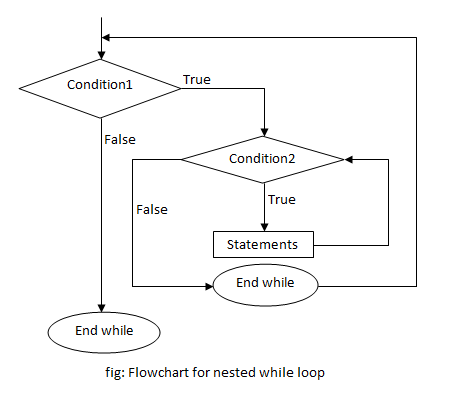
  ... ... ...

}

  ... ... ...

}

### Flowchart of Nested while loop



### Example of Nested while loop

**Example 1:** C program to print the number pattern.

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

#include <stdio.h>

int main()

{

int i=1,j;

while (i <= 5)

{

j=1;

while (j <= i )

{

printf("%d ",j);

j++;

}

printf("\n");

i++;

}

return 0;

}

In this program, nested while loop is used to print the pattern. The outermost loop runs 5 times and for every loop, the innermost loop runs i times which is 1 at first, meaning only "1" is printed, then on the next loop it's 2 numbers printing "1 2" and so on till 5 iterations of the loop executes, printing "1 2 3 4 5". This way, the given number pattern is printed.

## Nested do-while loop

A do-while loop inside another do-while loop is called nested do-while loop.

### Syntax of Nested do-while loop

do

{

  statement(s);

  do

{

  statement(s);

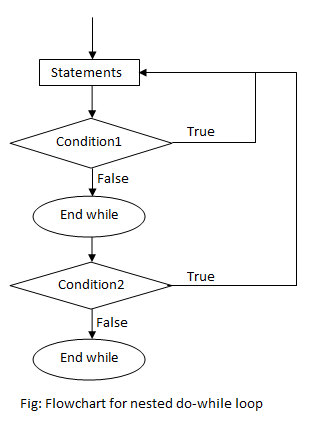
  ... ... ...

}while (condition2);

... ... ...

}while (condition1);

### Flowchart of Nested do-while loop



### Example of Nested do-while loop

**Example 2:** C program to print the given star pattern.

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

#include <stdio.h>

int main()

{

int i=1,j;

do

{

j=1;

do

{

printf("\*");

j++;

}while(j <= i);

i++;

printf("\n");

}while(i <= 5);

return 0;

}

In this program, nested do-while loop is used to print the star pattern. The outermost loop runs 5 times and for every loop, the innermost loop runs i times which is 1 at first, meaning only one "\*" is printed, then on the next loop it's 2 printing two stars and so on till 5 iterations of the loop executes, printing five stars. This way, the given star pattern is printed.

## Nested for loop

A for loop inside another for loop is called nested for loop.

### Syntax of Nested for loop

for (initialization; condition; increment/decrement)

{

  statement(s);

  for (initialization; condition; increment/decrement)

{

  statement(s);

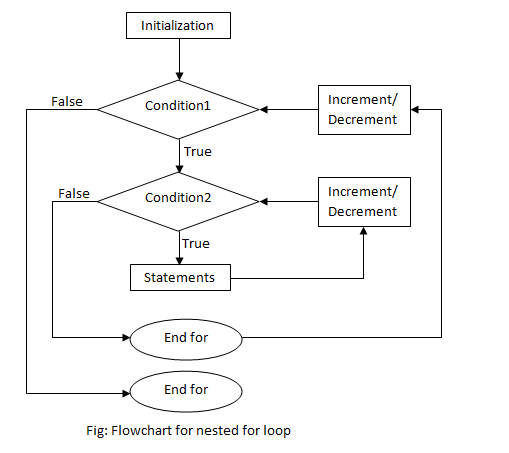
  ... ... ...

}

  ... ... ...

}

### Flowchart of Nested for loop



### Example of Nested for loop

**Example 3:** C program to print all the composite numbers from 2 to a certain number entered by user.

#include<stdio.h>

#include<math.h>

int main()

{

int i,j,n;

printf("Enter a number:");

scanf("%d",&n);

for(i=2;i<=n;i++)

{

for(j=2;j<=(int)pow(i,0.5);j++)

{

if(i%j==0)

{

printf("%d is composite\n",i);

break;

}

}

}

return 0;

}

Function

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition**provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

## Defining a Function

The general form of a function definition in C programming language is as follows −

return\_type function\_name( parameter list ) {

body of the function

}

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

* **Return Type** − A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
* **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body** − The function body contains a collection of statements that define what the function does.

## Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return\_type function\_name( parameter list );

For the above defined function max(), the function declaration is as follows

int max(int, int);

## Calling a Function

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

#include <stdio.h>

/\* function declaration \*/

int max(int num1, int num2);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = *max(a, b);*

printf( "Max value is : %d\n", ret );

return 0;

}

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

## 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.

|  |  |
| --- | --- |
| **Sr.No.** | **Call Type & Description** |
| 1 | [**Call by value**](https://www.tutorialspoint.com/cprogramming/c_function_call_by_value.htm)  This method 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. |
| 2 | [**Call by reference**](https://www.tutorialspoint.com/cprogramming/c_function_call_by_reference.htm)  This method 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. This means that changes made to the parameter affect the argument. |

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.

#include <stdio.h>

/\* function declaration \*/

void swap(int x, int y);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

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

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

/\* calling a function to swap the values \*/

swap(a, b);

printf("After swap, value of a : %d\n", a );

printf("After swap, value of b : %d\n", b );

return 0;

}

Call by refferance

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.

#include <stdio.h>

/\* function declaration \*/

void swap(int \*x, int \*y);

int main () {

/\* local variable definition \*/

int a = 100;

int b = 200;

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

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

/\* calling a function to swap the values.

\* &a indicates pointer to a ie. address of variable a and

\* &b indicates pointer to b ie. address of variable b.

\*/

swap(&a, &b);

printf("After swap, value of a : %d\n", a );

printf("After swap, value of b : %d\n", b );

return 0;

}

Scope and lifetime of variables

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable 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.

Let us understand what are **local** and **global** variables, and **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. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

[Live Demo](http://tpcg.io/HjGz74)

#include <stdio.h>

int main () {

/\* local variable declaration \*/

int a, b;

int c;

/\* actual initialization \*/

a = 10;

b = 20;

c = a + b;

printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

return 0;

}

## Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

[Live Demo](http://tpcg.io/TkoIHt)

#include <stdio.h>

/\* global variable declaration \*/

int g;

int main () {

/\* local variable declaration \*/

int a, b;

/\* actual initialization \*/

a = 10;

b = 20;

g = a + b;

printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

return 0;

}

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example −

[Live Demo](http://tpcg.io/Nx9joC)

#include <stdio.h>

/\* global variable declaration \*/

int g = 20;

int main () {

/\* local variable declaration \*/

int g = 10;

printf ("value of g = %d\n", g);

return 0;

}

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

value of g = 10

## Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example −

[Live Demo](http://tpcg.io/bcdMrO)

#include <stdio.h>

/\* global variable declaration \*/

int a = 20;

int main () {

/\* local variable declaration in main function \*/

int a = 10;

int b = 20;

int c = 0;

printf ("value of a in main() = %d\n", a);

c = sum( a, b);

printf ("value of c in main() = %d\n", c);

return 0;

}

/\* function to add two integers \*/

int sum(int a, int b) {

printf ("value of a in sum() = %d\n", a);

printf ("value of b in sum() = %d\n", b);

return a + b;

}

## Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you 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 your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

# Types of Functions in C

In real time, a [function](https://www.tutorialgateway.org/functions-in-c/) may be defined with or without parameters and a function may or may not return a value. It completely depends upon the user requirement.

The following are list of available types of Functions in C

1. Function with no argument and no Return value
2. Function with no argument and with a Return value
3. Function with argument and No Return value
4. Function with argument and Return value

### 1.C Function with No argument and No Return value

In this method, We won’t pass any arguments to the function while defining, declaring or calling the function. This type of functions in C will not return any value when we call the function from main() or any sub function. When we are not expecting any return value but, we need some statements to be printed as output then, this type of functions in C are very useful.

#### C Function with No argument and No Return value Example

In this program, We are going to calculate the Sum of 2 integer values and print the output from the user defined function itself.

/\* Function with No argument and No Return value Example \*/

#include<stdio.h>

// Function Declaration

void Addition();

void main()

{

printf("\n ............. \n");

Addition();

}

void Addition()

{

int Sum, a = 10, b = 20;

Sum = a + b;

printf("\n Sum of a = %d and b = %d is = %d", a, b, Sum);

}

### 2.C Function with no argument and with Return value

In this method, We won’t pass any arguments to the function while defining, declaring or calling the function. This type of functions will return some value when we call the function from main() or any subfunction.

The Data Type of the return value will depend upon the return type of function declaration. For instance, if the return type is int then return value will be int.

#### Function with No arguments and with Return value Example

In this program, We are going to calculate the multiplication of 2 integer values using the user defined function without arguments and return keyword.

#include<stdio.h>

int Multiplication();

int main()

{

int Multi;

Multi = Multiplication();

printf("\n Multiplication of a and b is = %d \n", Multi );

return 0;

}

int Multiplication()

{

int Multi, a = 20, b = 40;

Multi = a \* b;

return Multi;

}

### 3. C Function with argument and No Return value

If you observe the above 2 methods, No matter how many times you executive, it will give the same output. We don’t have any control over the values of the variables a and b because they are fixed values.

In real time, we mostly deal with dynamic data means we have to allow the user to enter his own values rather than fixed ones.

This method allows us to pass the arguments to the function while calling the function. But, This type of functions will not return any value when we call the function from main () or any subfunction.

If we want to allow our user to pass his own data to the function arguments but we are not expecting any return value then, this type of functions are very useful.

#### Function with argument and No Return value Example

This program allows the user to enter 2 integer values and then, We are going to pass those values to the user-defined function to calculate the sum.

#include<stdio.h>

void Addition(int, int);

void main()

{

int a, b;

printf("\n Please Enter two integer values \n");

scanf("%d %d",&a, &b);

//Calling the function with dynamic values

Addition(a, b);

}

void Addition(int a, int b)

{

int Sum;

Sum = a + b;

printf("\n Additiontion of %d and %d is = %d \n", a, b, Sum);

}

### 4. C Function with argument and Return value

This method allows us to pass the arguments to the function while calling the function. This type of functions will return some value when we call the function from main () or any subfunction. Data Type of the return value will depend upon the return type of function declaration. For instance, if the return type is int then return value will be int.

This type of user-defined functions are called as a fully dynamic function means, it provides maximum control to the end user.

#### Function with arguments and Return value Example

This program allows the user to enter 2 integer values and then, We are going to pass those values to the user-defined function to multiply those values and return the value using return keyword.

#include<stdio.h>

int Multiplication(int, int);

int main()

{

int a, b, Multi;

printf("\n Please Enter two integer values \n");

scanf("%d %d",&a, &b);

//Calling the function with dynamic values

Multi = Multiplication(a, b);

printf("\n Multiplication of %d and %d is = %d \n", a, b, Multi);

return 0;

}

int Multiplication(int a, int b)

{

int Multi;

Multi = a \* b;

return Multi;

}

Recursion

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

void recursion() {

recursion(); /\* function calls itself \*/

}

int main() {

recursion();

}

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.

## Number Factorial

The following example calculates the factorial of a given number using a recursive function −

[Live Demo](http://tpcg.io/os8QlL)

#include <stdio.h>

unsigned long long int factorial(unsigned int i) {

if(i <= 1) {

return 1;

}

return i \* factorial(i - 1);

}

int main() {

int i = 12;

printf("Factorial of %d is %d\n", i, factorial(i));

return 0;

}

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

Factorial of 12 is 479001600

## Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function −

[Live Demo](http://tpcg.io/GsweWK)

#include <stdio.h>

int fibonacci(int i) {

if(i == 0) {

return 0;

}

if(i == 1) {

return 1;

}

return fibonacci(i-1) + fibonacci(i-2);

}

int main() {

int i;

for (i = 0; i < 10; i++) {

printf("%d\t\n", fibonacci(i));

}

return 0;

}

Array

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



## Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [ arraySize ];

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

double balance[10];

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

## Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

balance[4] = 50.0;

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −



## Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays −

[Live Demo](http://tpcg.io/dhfplr)

#include <stdio.h>

int main () {

int n[ 10 ]; /\* n is an array of 10 integers \*/

int i,j;

/\* initialize elements of array n to 0 \*/

for ( i = 0; i < 10; i++ ) {

n[ i ] = i + 100; /\* set element at location i to i + 100 \*/

}

/\* output each array element's value \*/

for (j = 0; j < 10; j++ ) {

printf("Element[%d] = %d\n", j, n[j] );

}

return 0;

}

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

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

Two dimension array

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration −

type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional integer array −

int threedim[5][10][4];

## Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

type arrayName [ x ][ y ];

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows −



Thus, every element in the array **a** is identified by an element name of the form **a[ i ][ j ]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

## Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

int a[3][4] = {

{0, 1, 2, 3} , /\* initializers for row indexed by 0 \*/

{4, 5, 6, 7} , /\* initializers for row indexed by 1 \*/

{8, 9, 10, 11} /\* initializers for row indexed by 2 \*/

};

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};

## Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

int val = a[2][3];

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array −

[Live Demo](http://tpcg.io/uZJ0tv)

#include <stdio.h>

int main () {

/\* an array with 5 rows and 2 columns\*/

int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

int i, j;

/\* output each array element's value \*/

for ( i = 0; i < 5; i++ ) {

for ( j = 0; j < 2; j++ ) {

printf("a[%d][%d] = %d\n", i,j, a[i][j] );

}

}

return 0;

}

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

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Array as argument to the function

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

## Way-1

Formal parameters as a pointer −

void myFunction(int \*param) {

.

.

.

}

## Way-2

Formal parameters as a sized array −

void myFunction(int param[10]) {

.

.

.

}

## Way-3

Formal parameters as an unsized array −

void myFunction(int param[]) {

.

.

.

}

## Example

Now, consider the following function, which 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 as follows −

double getAverage(int arr[], int size) {

int i;

double avg;

double sum = 0;

for (i = 0; i < size; ++i) {

sum += arr[i];

}

avg = sum / size;

return avg;

}

Now, let us call the above function as follows −

#include <stdio.h>

/\* function declaration \*/

double getAverage(int arr[], int size);

int main () {

/\* an int array with 5 elements \*/

int balance[5] = {1000, 2, 3, 17, 50};

double avg;

/\* pass pointer to the array as an argument \*/

avg = getAverage( balance, 5 ) ;

/\* output the returned value \*/

printf( "Average value is: %f ", avg );

return 0;

}

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

Average value is: 214.400000

As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for formal parameters.

String

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++ −



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

[Live Demo](http://tpcg.io/P0muN5)

#include <stdio.h>

int main () {

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

printf("Greeting message: %s\n", greeting );

return 0;

}

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

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings −

|  |  |
| --- | --- |
| **Sr.No.** | **Function & Purpose** |
| 1 | **strcpy(s1, s2);**  Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);**  Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);**  Returns the length of string s1. |
| 4 | **strcmp(s1, s2);**  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);**  Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);**  Returns a pointer to the first occurrence of string s2 in string s1. |

The following example uses some of the above-mentioned functions −

[Live Demo](http://tpcg.io/VBA8Qx)

#include <stdio.h>

#include <string.h>

int main () {

char str1[12] = "Hello";

char str2[12] = "World";

char str3[12];

int len ;

/\* copy str1 into str3 \*/

strcpy(str3, str1);

printf("strcpy( str3, str1) : %s\n", str3 );

/\* concatenates str1 and str2 \*/

strcat( str1, str2);

printf("strcat( str1, str2): %s\n", str1 );

/\* total lenghth of str1 after concatenation \*/

len = strlen(str1);

printf("strlen(str1) : %d\n", len );

return 0;

}

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

strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10