

Programming in C

Module: 1:

Introduction to C Language.

Introduction to Programming – Algorithms – Pseudo Code - Flow Chart – Compilation – Execution – Preprocessor Directives (#define, #include, #undef) - Overview of C – Constants, Variables and Data types – Operators and Expressions – Managing Input and Output Operations – Decision Making and Branching - Decision Making and Looping.

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

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 the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

It is a very powerful and general-purpose language used in programming. We can use C to develop software such as databases, operating systems, compilers, and many more. This programming language is excellent to learn for beginners in programming.

C language is considered as the mother language of all the modern programming languages because **most of the compilers, JVMs, Kernels, etc. are written in C language**, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

A system programming language is used to create system software. C language is a system programming language because it **can be used to do low-level programming (for example driver and kernel)**. It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem**.

A procedural language breaks the program into functions, data structures, etc.

A structured programming language is a subset of the procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

C is considered as a middle-level language because it **supports the feature of both low-level and high-level languages**. C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

How to install C

There are many compilers available for c and c++. You need to download any one. Here, we are going to use **Turbo C++**. It will work for both C and C++. To install the Turbo C software, you need to follow following steps.

1. Download Turbo C++
2. Create turboc directory inside c drive and extract the tc3.zip inside c:\turboc
3. Double click on install.exe file
4. Click on the tc application file located inside c:\TC\BIN to write the c program

Download links:

1. <https://developerinsider.co/download-turbo-c-for-windows-7-8-8-1-and-windows-10-32-64-bit-full-screen/>
2. <https://static.javatpoint.com/cpages/software/tc3.zip>

Algorithm:

An algorithm is a step-by-step procedure or a set of rules designed to perform a specific task or solve a problem. It provides a clear and logical sequence of actions to achieve a particular goal.

Pseudocode

Pseudocode is a way to describe an algorithm using a mix of natural language and programming-like syntax. It doesn't follow any specific programming language but resembles code to help in understanding how the algorithm will be implemented in a real programming language.

Key Differences

1. **Nature:**
 - **Algorithm:** A list of steps or rules.
 - **Pseudocode:** A textual description, close to code but not in a specific programming language.
 - **Flowchart:** A visual diagram showing the sequence of steps and decisions.
2. **Purpose:**
 - **Algorithm:** To describe the process clearly and logically.
 - **Pseudocode:** To help in the transition from an algorithm to actual programming code.
 - **Flowchart:** To provide a visual overview of the process, making it easier to understand.
3. **Representation:**
 - **Algorithm:** Written in plain language or structured format.
 - **Pseudocode:** Written in a style similar to programming language syntax.
 - **Flowchart:** Drawn using symbols and arrows.

Example 1: Adding Two Numbers

Algorithm

1. Start.
2. Input two numbers (let's call them A and B).
3. Calculate the sum: $\text{Sum} = A + B$.
4. Display the result (Sum).
5. End.






Pseudocode

```
Start
    Input A
    Input B
    Sum = A + B
    Display Sum
End
```

Flowchart

A flowchart is a visual representation of an algorithm. It uses various symbols like rectangles (for processes), diamonds (for decisions), and arrows to show the flow of steps. Flowcharts help in understanding the sequence of actions and decisions involved in a process.

Guidelines for drawing a flowchart

| | |
|---|---|
|  | Start or end of the program |
|  | Computational steps or processing function of a program |
|  | Input or output operation |
|  | Decision making and branching |
|  | Connector or joining of two parts of program |

These examples show the structured approach to solving basic problems using algorithms and pseudocode, making the steps clear and easy to translate into any programming language

Key points:

Algorithm is the foundational idea.

Pseudocode bridges the gap between the algorithm and actual code.

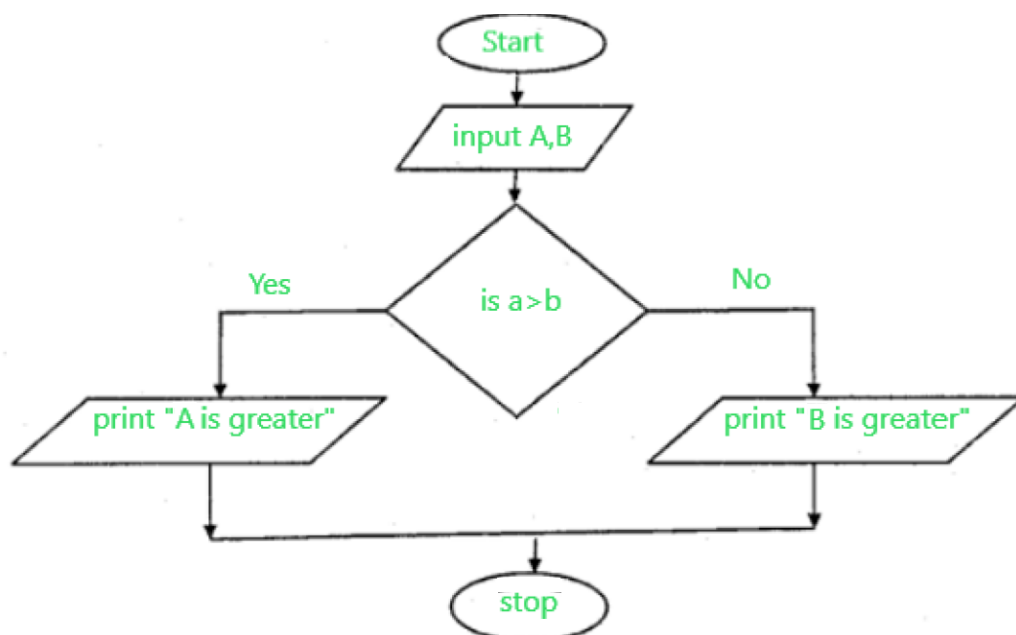
Flowchart visually represents the algorithm for better clarity and communication.

Example:

Develop a algorithm and draw a flow chart to find the greatest number among the 2 numbers.

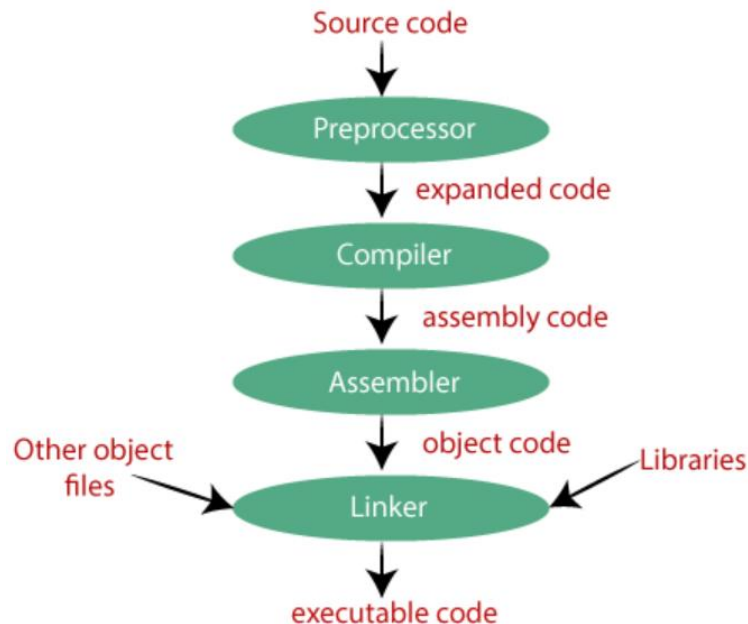
Algorithm:

1. Start
2. Input 2 variables from user
3. Now check the condition If $a > b$, goto step 4, else goto step 5.
4. Print a is greater, goto step 6
5. Print b is greater
6. Stop

Flow Chart:**Compilation process in C Programming**

The compilation process has four different steps –

- Preprocessing
- Compiling
- Assembling
- Linking



Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'. If the name of the source file is '**hello.c**', then the name of the object file would be 'hello.obj'.

Linker

Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.h') extension. The main working of the linker is to combine the object code of library files with the object code of our program. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is the executable file.

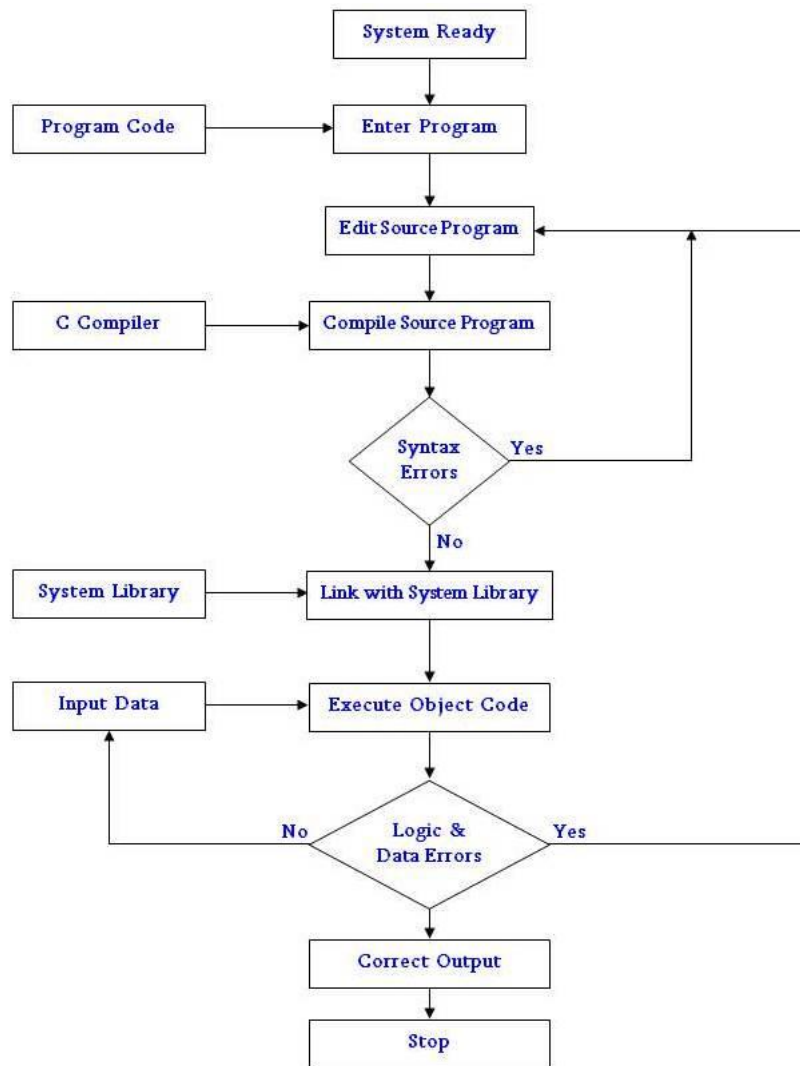


Figure : Process of compiling and running a C program.

In C language, the **Preprocessor** is a tool that processes the source code before it is compiled by the compiler. It handles directives that begin with the # symbol and performs operations such as including files, defining constants, and conditional compilation. The preprocessor operates on the source code and modifies it, but it does not compile the code itself.

Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Main Role of the Preprocessor:

The primary role of the preprocessor is to prepare the source code for the compiler. This involves several key tasks:

1. **File Inclusion (#include):**

- The preprocessor includes the content of other files (like header files) into the source file. For example, #include <stdio.h> includes the standard I/O library functions.

2. **Macro Substitution (#define):**

- It replaces macros with their defined values or code snippets throughout the source file. For instance, `#define PI 3.14` replaces every occurrence of `PI` with `3.14`.
- 3. **Conditional Compilation (`#ifdef`, `#ifndef`, `#endif`):**
 - It allows compiling parts of the code conditionally based on whether certain conditions are met. This is useful for platform-specific code or debugging.
- 4. **Removing Comments:**
 - The preprocessor removes all the comments from the source code, which are not needed in the compiled program.
- 5. **Other Directives:**
 - Directives like `#undef`, `#pragma`, and `#error` also control various aspects of the code compilation.

C Variable, Datatypes, Constants

What is a Variable?

A variable is an identifier which is used to store some value. Constants can never change at the time of execution. Variables can change during the execution of a program and update the value stored inside it.

A single variable can be used at multiple locations in a program. A variable name must be meaningful. It should represent the purpose of the variable.

Following are the rules that must be followed while creating a variable:

1. A variable name should consist of only characters, digits and an underscore.
2. A variable name should not begin with a number.
3. A variable name should not consist of whitespace.
4. A variable name should not consist of a keyword.
5. 'C' is a case sensitive language that means a variable named 'age' and 'AGE' are different.
6. Following are the examples of valid variable names in a 'C' program:
7. height or HEIGHT
8. _height
9. _height1
10. My_name
11. Following are the examples of invalid variable names in a 'C' program:
12. 1height
13. Hei\$ght
14. My name

Data types

'C' provides various data types to make it easy for a programmer to select a suitable data type as per the requirements of an application. Following are the three data types:

1. Primitive data types
2. Derived data types
3. User-defined data types

There are five primary fundamental data types,

1. int for integer data
2. char for character data

3. float for floating point numbers
4. double for double precision floating point numbers
5. void

| Data type | Size in bytes | Range |
|--|---------------|---------------------------|
| Char or signed char | 1 | -128 to 127 |
| Unsigned char | 1 | 0 to 255 |
| int or signed int | 2 | -32768 to 32767 |
| Unsigned int | 2 | 0 to 65535 |
| Short int or Unsigned short int | 2 | 0 to 255 |
| Signed short int | 2 | -128 to 127 |
| Long int or Signed long int | 4 | -2147483648 to 2147483647 |
| Unsigned long int | 4 | 0 to 4294967295 |
| 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 |

Integer data type

Integer is nothing but a whole number. The range for an integer data type varies from machine to machine. The standard range for an integer data type is -32768 to 32767.

An integer typically is of 2 bytes which means it consumes a total of 16 bits in memory. A single integer value takes 2 bytes of memory. An integer data type is further divided into other data types such as short int, int, and long int.

Each data type differs in range even though it belongs to the integer data type family. The size may not change for each data type of integer family.

The short int is mostly used for storing small numbers, int is used for storing averagely sized integer values, and long int is used for storing large integer values.

Whenever we want to use an integer data type, we have place int before the identifier such as,

```
int age;
```

Here, age is a variable of an integer data type which can be used to store integer values.

Floating point data type

The 'float' keyword is used to represent the floating point data type. It can hold a floating point value which means a number is having a fraction and a decimal part. A floating point value is a real number that contains a decimal point. Integer data type doesn't store the decimal part hence we can use floats to store decimal part of a value.

Generally, a float can hold up to 6 precision values. If the float is not sufficient, then we can make use of other data types that can hold large floating point values. The data type double and long double are used to store real numbers with precision up to 14 and 80 bits respectively.

While using a floating point number a keyword float/double/long double must be placed before an identifier. The valid examples are,

```
float division;
```

```
double BankBalance;
```

Character data type

Character data types are used to store a single character value enclosed in single quotes.

A character data type takes up-to 1 byte of memory space.

Example:

```
Char letter;
```

Void data type

A void data type doesn't contain or return any value. It is mostly used for defining functions in 'C'.

Example:

```
void displayData()
```

Type declaration of a variable

```
void main() {  
int x, y;  
float salary = 13.48;  
char letter = 'K';  
x = 25;  
y = 34;  
int z = x+y;  
printf("%d \n", z);  
printf("%f \n", salary);  
printf("%c \n", letter);  
}
```

Output:

```
59  
13.480000  
K
```

We can declare multiple variables with the same data type on a single line by separating them with a comma. Also, notice the use of format specifiers in **printf** output function float (%f) and char (%c) and int (%d).

Constants

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

Integer constants

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

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

Example, 111, 1234

Above are the valid decimal constants.

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

Example, 012, 065

Above are the valid Octal constants.

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

Example, 0X2, 0Xbcd

Above are the valid hexadecimal constants.

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

Character constants

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

Example, 'A', '9'

Above are the examples of valid character constants.

String constants

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

Example, "Hello", "Programming"

These are the examples of valid string constants.

Real Constants

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

Example, 202.15, 300.00

These are the valid real constants in 'C'.

A real constant can also be written as,

Mantissa e Exponent

For example, to declare a value that does not change like the classic circle constant PI, there are two ways to declare this constant

1. By using the **const** keyword in a variable declaration which will reserve a storage memory

```
#include <stdio.h>
void main() {
    const double PI = 3.14;
    printf("%f", PI);
    //PI++; // This will generate an error as constants cannot be changed
}
```

2. By using the **#define** pre-processor directive which doesn't use memory for storage and without putting a semicolon character at the end of that statement

```
#include <stdio.h>
#define PI 3.14
void main() {
    printf("%f", PI);
}
```

Key Points

- A constant is a value that doesn't change throughout the execution of a program.
- A variable is an identifier which is used to store a value.
- There are four commonly used data types such as int, float, char and a void.
- Each data type differs in size and range from one another.

In this Section, you will learn about Operators in C Programming (all valid operators available in C), expressions (combination of operators, variables and constants) and precedence of operators (which operator has higher priority and which operator has lower priority).

1. C Operators
2. Expressions in C
3. C Operator Precedence
- 4.

C Operators

Operators are the symbols which tell the computer to execute certain mathematical or logical operations. A mathematical or logical expression is generally formed with the help of an operator. C programming offers a number of operators which are classified into 8 categories viz.

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

1. Arithmetic Operators

C programming language provides all basic arithmetic operators: +, -, *, / and %.

| Operators | Meanings |
|-----------|----------------------------|
| + | Addition or unary plus |
| - | Subtraction or unary minus |
| * | Multiplication |
| / | Division |
| % | Modulo division |

Table 1: Arithmetic Operators in C

Note: '/' is integer division which only gives integer part as result after division. '%' is modulo division which gives the remainder of integer division as result.

Some examples of arithmetic operators are:

- 1) $a + b$ 2) $a - b$ 3) $a * b$ 4) a / b 5) $a \% b$

In these examples, a and b are variables and are called operands.

Note: '%' cannot be used on floating data type.

2. Relational Operators

Relational operators are used when we have to make comparisons. C programming offers 6 relational operators.

| Operators | Meanings |
|-----------|-----------------------------|
| < | Is less than |
| <= | Is less than or equal to |
| > | Is greater than |
| >= | Is greater than or equal to |
| == | Is equal to |
| != | Is not equal to |

Table 2: Relational Operators in C

Relational expression is an expression which contains the relational operator. Relational operators are most commonly used in decision statements like *if*, *while*, etc. Some simple relational expressions are:

- 1 < 5

- $9 \neq 8$
- $2 > 1+3$

Note: Arithmetic operators have higher priority than relational operators meaning that if arithmetic expressions are present on two sides of a relational operator then arithmetic expressions will be calculated first and then the result will be compared.

3. Logical Operators

Logical operators are used when more than one conditions are to be tested and based on that result, decisions have to be made. C programming offers three logical operators. They are:

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

Table 3: Logical Operator in C

For example:

$a < 18 \text{ || } a > 60$

An expression which combines two or more relational expressions is known as logical expression.

Note: Relative precedence of relational and logical operators are as follows

| | |
|--------------------|-----------|
| Highest precedence | ! |
| | > >= < <= |
| | == != |
| | && |
| Lowest precedence | |

4. Assignment Operators

Assignment operators are used to assign result of an expression to a variable. '=' is the assignment operator in C. Furthermore, C also allows the use of shorthand assignment operators. Shorthand operators take the form:

```
var op = exp;
```

where *var* is a variable, *op* is arithmetic operator, *exp* is an expression. In this case, 'op=' is known as shorthand assignment operator.

The above assignment

```
var op = exp;
```

is the same as the assignment

```
var = var op exp;
```

Consider an example:

```
x += y;
```

Here, the above statement means the same as

```
x = x + y;
```

Note: Shorthand assignment can be used with all arithmetic operators.

5. Increment and Decrement Operators

C programming allows the use of ++ and – operators which are increment and decrement operators respectively. Both the increment and decrement operators are unary operators. The increment operator ++ adds 1 to the operand and the decrement operator – subtracts 1 from the operand. The general syntax of these operators are:

Increment Operator: *m++ or ++m;*

Decrement Operator: *m–or –m;*

In the example above, *m++* simply means *m=m+1;* and *m–* simply means *m=m-1;*

Increment and decrement operators are mostly used in for and while loops.

++m and *m++* performs the same operation when they form statements independently but they function differently when they are used in right hand side of an expression.

++m is known as prefix operator and *m++* is known as postfix operator. A prefix operator firstly adds 1 to the operand and then the result is assigned to the variable on the left whereas a postfix operator firstly

assigns value to the variable on the left and then increases the operand by 1. Same is in the case of decrement operator.

For example,

```
X=10;  
Y=++X;
```

In this case, the value of X and Y will be 6.

And,

```
X=10;  
  
Y=X++;
```

In this case, the value of Y will be 10 and the value of X will be 11.

6. Conditional Operator

The operator pair "?" and ":" is known as conditional operator. These pair of operators are ternary operators. The general syntax of conditional operator is:

expression1 ? expression2 : expression3 ;

This syntax can be understood as a substitute of if else statement.

For example,

```
a = 3 ;  
  
b = 5 ;
```

Consider an if else statement as:

```
if (a > b)
```

```
  x = a ;
```

```
else
```

```
  x = b ;
```

Now, this if else statement can be written by using conditional operator as:

```
x = (a > b) ? a : b ;
```


7. Bitwise Operator

In C programming, bitwise operators are used for testing the bits or shifting them left or right. The bitwise operators available in C are:

| Operators | Meaning |
|-----------|----------------------|
| & | Bitwise AND |
| ! | Bitwise OR |
| ^ | Bitwise exclusive OR |
| << | Shift left |
| >> | Shift right |

Table 4: Bitwise Operators in C

8. Special Operators

C programming supports special operators like comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. and ->). The comma operator and sizeof operator are discussed in this section whereas the pointer and member selection operators are discussed in later sections.

Comma Operator

The comma operator can be used to link the related expressions together. A comma linked expression is evaluated from left to right and the value of the right most expression is the value of the combined expression.

For example:

```
x = (a = 2, b = 4, a+b)
```

In this example, the expression is evaluated from left to right. So at first, variable a is assigned value 2, then variable b is assigned value 4 and then value 6 is assigned to the variable x. Comma operators are commonly used in for loops, while loops, while exchanging values, etc.

Sizeof() operator

The sizeof operator is usually used with an operand which may be variable, constant or a data type qualifier. This operator returns the number of bytes the operand occupies. Sizeof operator is a compile time operator. Some examples of use of sizeof operator are:

```
x = sizeof (a);
```

```
y = sizeof(float);
```

The sizeof operator is usually used to determine the length of arrays and structures when their sizes are not known. It is also used in dynamic memory allocation.

9. C Expressions

Arithmetic expression in C is a combination of variables, constants and operators written in a proper syntax. C can easily handle any complex mathematical expressions but these mathematical expressions have to be written in a proper syntax. Some examples of mathematical expressions written in proper syntax of C are:

Note: C does not have any operator for exponentiation.

10.C Operator Precedence

At first, the expressions within parenthesis are evaluated. If no parenthesis is present, then the arithmetic expression is evaluated from left to right. There are two priority levels of operators in C.

High priority: * / %

Low priority: + -

The evaluation procedure of an arithmetic expression includes two left to right passes through the entire expression. In the first pass, the high priority operators are applied as they are encountered and in the second pass, low priority operations are applied as they are encountered.

Suppose, we have an arithmetic expression as:

$$x = 9 - 12 / 3 + 3 * 2 - 1$$

This expression is evaluated in two left to right passes as:

First Pass

Step 1: $x = 9 - 4 + 3 * 2 - 1$

Step 2: $x = 9 - 4 + 6 - 1$

Second Pass

Step 1: $x = 5 + 6 - 1$

Step 2: $x = 11 - 1$

Step 3: $x = 10$

But when parenthesis is used in the same expression, the order of evaluation gets changed.

For example,

$$x = 9 - 12 / (3 + 3) * (2 - 1)$$

When parentheses are present then the expression inside the parenthesis are evaluated first from left to right. The expression is now evaluated in three passes as:

First Pass

Step 1: $x = 9 - 12 / 6 * (2 - 1)$

Step 2: $x = 9 - 12 / 6 * 1$

Second Pass

Step 1: $x = 9 - 2 * 1$

Step 2: $x = 9 - 2$

Third Pass

Step 3: $x = 7$

There may even arise a case where nested parentheses are present (i.e. parenthesis inside parenthesis). In such case, the expression inside the innermost set of parentheses is evaluated first and then the outer parentheses are evaluated.

For example, we have an expression as:

$$x = 9 - ((12 / 3) + 3 * 2) - 1$$

The expression is now evaluated as:

First Pass:

Step 1: $x = 9 - (4 + 3 * 2) - 1$

Step 2: $x = 9 - (4 + 6) - 1$

Step 3: $x = 9 - 10 - 1$

Second Pass

Step 1: $x = -1 - 1$

Step 2: $x = -2$

Note: The number of evaluation steps is equal to the number of operators in the arithmetic expression.

Managing Input and Output Operations

Types of Input and Output Functions

We have the following categories of IO function in C –

- **Unformatted character IO functions:** getchar() and putchar()
- **Unformatted string IO functions:** gets() and puts()
- **Formatted IO functions:** scanf() and printf()

The unformatted I/O functions read and write data as a stream of bytes without any format, whereas formatted I/O functions use predefined formats like "%d", "%c" or "%s" to read and write data from a stream.

| Standard File | File | Device |
|-----------------|--------|-------------|
| Standard input | stdin | Keyboard |
| Standard output | stdout | Screen |
| Standard error | stderr | Your screen |

Unformatted Character Input & Output Functions: getchar() and putchar()

These two functions accept a single character as input from the keyboard, and display a single character on the output terminal, respectively.

The getchar() function it reads a single key stroke without the Enter key.

int getchar(void)

There are no parameters required. The function returns an integer corresponding to the ASCII value of the character key input by the user.

Example 1

The following program reads a single key into a char variable –

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

```
void main() {
```

```
    char ch;
```

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

```
    ch = getchar();
```

```
    puts("You entered: ");
```

```
putchar(ch);
```

```
}
```

Output

Run the code and check its output –

Enter a character: W

You entered: W

You can also use the unformatted `putchar()` function to print a single character. The C library function "int `putchar(int char)`" writes a character (an unsigned char) specified by the argument `char` to `stdout`.

```
int putchar(int c)
```

A single parameter to this function is the character to be written. You can also pass its ASCII equivalent integer. This function returns the character written as an unsigned char cast to an int or EOF on error.

Formatted Input & Output Functions: `scanf()` and `printf()`

The `scanf()` function reads the input from the standard input stream `stdin` and scans that input according to the format provided –

```
int scanf(const char *format, ...)
```

The `printf()` function writes the output to the standard output stream `stdout` and produces the output according to the format provided.

```
int printf(const char *format, ...)
```

The format can be a simple constant string, but you can specify `%s`, `%d`, `%c`, `%f`, etc., to print or read strings, integers, characters or floats respectively. There are many other formatting options available which can be used based on the specific requirements.

Format Specifiers in C

The CPU performs IO operations with input and output devices in a streaming manner. The data read from a standard input device (keyboard) through the standard input stream is called `stdin`. Similarly, the data sent to the standard output (computer display screen) through the standard output device is called `stdout`.

The computer receives data from the stream in a text form, however you may want to parse it in variables of different data types such as `int`, `float` or a string. Similarly, the data stored in `int`, `float` or `char` variables has to be sent to the output stream in a text format. The format specifier symbols are used exactly for this purpose.

ANSI C defines a number of format specifiers. The following table lists the different specifiers and their purpose.

| Format Specifier | Type |
|------------------|-------------------------------|
| %c | Character |
| %d | Signed integer |
| %f | Float values |
| %s | String |
| %hi | Signed integer (short) |
| %hu | Unsigned Integer (short) |
| %i | Unsigned integer |
| %l or %ld or %li | Long |
| %lf | Double |
| %Lf | Long double |
| %lu | Unsigned int or unsigned long |
| %lli or %lld | Long long |
| %llu | Unsigned long long |
| %o | Octal representation |
| %p | Pointer |
| %u | Unsigned int |
| %x or %X | Hexadecimal representation |

Decision Making and Branching - Decision Making and Looping

This article will discuss "Decision Making Branching Looping in C program".

Decision Making: This involves using conditions to make choices in code. For example, using an "if" statement to check if a certain condition is true or false, and then executing different blocks of code accordingly.

Branching: Branching is an extension of decision making. It involves creating multiple paths or branches in your code based on different conditions. This often includes using "else if" statements.

Looping: Looping allows you to repeat a certain block of code multiple times. There are different types of loops, but two common ones are "for" and "while" loops.

Decision-Making statements

- It will change the execution order with the checking process.

Types

1. Simple **if Statements**
2. **if-else** statements
3. else ...if ladder statements
4. nested if statements
5. switch statements

Simple if Statements

It will change the execution order from one place to another place (or) skip the execution of a group of statements (or) Execute the group of statements by checking one condition.

Syntax :

if(Boolean_expression) goto label; (or)

if(Boolean_expression) statement; (or)

if(Boolean_expression)

```
{  
    statement_1;  
    statement_2;  
}
```

Note: statement(s) will execute if the Boolean expression is true.

Rules

Test conditions should be in brackets ()

Test condition should be Relational /Logical expression

Should use open{ and } close braces, if use group of statements

examples

if (a<b) big =1;

if (a=1) goto ArunEworld;

```
if (a<b)
```

```
{
```

```
    s=a+b;
```

```
    d =a-b;
```

```
    m = a+b;
```

```
}
```

Example Programs

Evaluate_mark_of_students_using_if_statement.c

Evaluvate_sale_tax_using_if_statement.c

if- else statements

Execute the group of statement if test condition is true, else test condition is false it will execute the another group of statements

Syntax :

```
    If(Boolean_expression)  statement_1;
```

```
    else
```

```
        statement_2;      (or)
```

```
if(Boolean_expression)
```

```
{
```

```
    statement_1;
```

```
    statement_2;
```

```
}
```

```
else
```

```
{
```

```
    statement_1;
```

```
    statement_2;
```

```
}
```


Rules

Test conditions should be in brackets ()

Test condition should be Relational /Logical expression

Should use open { and } close braces, if use group of statements

examples

```
if (a<b) big =b; else big =a;
```

```
if (a=1) goto label1; else goto label2;
```

```
if (a<b)
```

```
    { s=a+b; d =a-b; }
```

```
    else
```

```
        m = a+b;
```

Example Programs

check_given_marks_fail_or_pass_using_if_else_statement.c

Biggest_number_using_if_statement.c

else ...if ladder statements

else..if is can use to checking more than one conditions, Execute the group of statement if test condition_1 is true, else..if test condition_2 is true it will execute the another group of statements, else above two test condition is false execute another group of statements

Syntax :

```
If (Test_condition_1)
```

```
    statement_1;
```

```
elseif (Test_Condition_2)
```

```
    statement_2;
```

```
else
```

```
    statement_3;
```

```
(or)
```

```
if(Test_condition_1)

    {   statement_1;

        statement_2;

    }
```

Elseif (Test_condition_2)

```
{

statement_1;

statement_2;

}
```

else

```
{

statement_1;

statement_2;

}
```

Rules

Test conditions should be in brackets ()

Test condition should be Relational /Logical expression

Should use open{ and } close braces, if use group of statements

examples

if (a<b)

big =b;

elseif(b>a)

big = a;

else

printf("a and b are same vale");

if (a>b) goto label1; elseif(b>a) goto label2; else goto label1;

```
if (a<b)
```

```
{ s=a+b;d =a-b; }
```

```
Eslesif (b>a)
```

```
{ d =a-b; b= a-b;}
```

```
else
```

```
{m = a+b; a=a+b;}
```

Example Programs

Display_Day_using_if_elseif_statement.c

nested if statements

Syntax :

```
if( boolean_expression)
```

```
{
```

```
    /* Executes when the boolean expression 1 is true */
```

```
    if(boolean_expression 2)
```

```
    {
```

```
        /* Executes when the boolean expression 2 is true */
```

```
    }
```

```
}
```

Examples : Biggest_of_three_numbers_using_nested-if.c

Looping:

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language. In this part of the tutorial, we are going to learn all the aspects of C loops.

Why use loops in C language?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantage of loops in C

- 1) It provides code reusability.
- 2) Using loops, we do not need to write the same code again and again.
- 3) Using loops, we can traverse over the elements of data structures (array or linked lists).

Types of C Loops

There are three types of loops in C language that is given below:

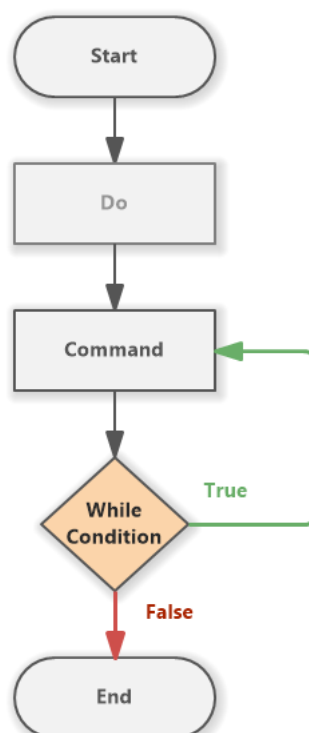
1. do while loop
2. while loop
3. for loop

The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

The syntax of do-while loop in c language is given below:

```
do{  
    //code to be executed  
} while(condition);
```

Flowchart and Example of do-while loop



A "do-while" loop is a form of a loop in C that executes the code block first, followed by the condition. If the condition is true, the loop continues to run; else, it stops. However, whether the condition is originally true, **it ensures that the code block is performed at least once.**

```
int i=1;

do

{

    printf("%d",i)

    i=i+1;

} while(i<=5);
```

Output:

```
1
2
3
4
5
```

while loop in C

The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

The syntax of while loop in c language is given below:

```
while(condition){

//code to be executed

}
```

Example:

```
int i=1;

while(i<=5)

{

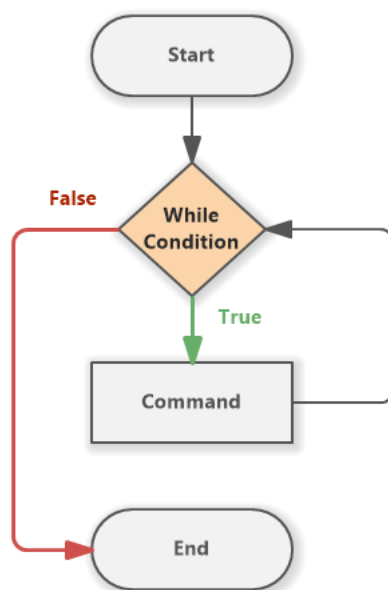
    printf("%d",i)

    i=i+1; }
```

Output:

1
2
3
4
5

Flowchart of do-while loop



for loop in C

The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a pre-tested loop. It is better to use for loop if the number of iteration is known in advance.

The syntax of for loop in c language is given below:

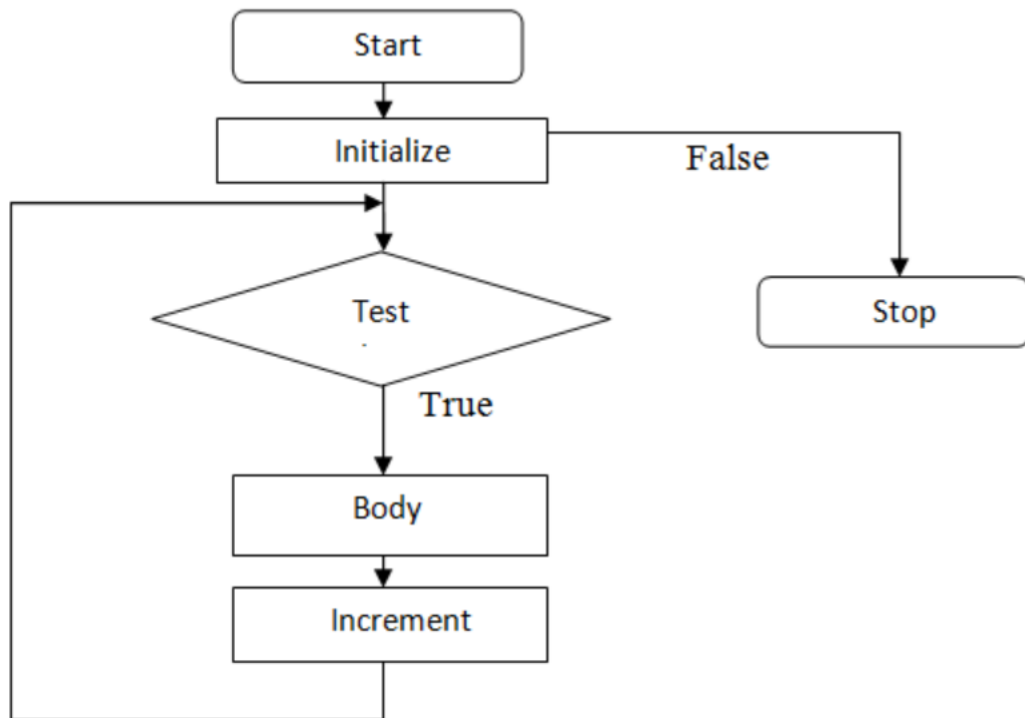
```
for(initialization; condition; increment/decrement)
```

```
{
```

```
//code to be executed
```

```
}
```

Flowchart of do-while loop



Let's see the simple program of for loop that prints table of 1.

```
1. #include<stdio.h>
2. void main(){
3. int i=0;
4. for(i=1;i<=10;i++){
5. printf("%d \n",i);
6. }
7.
8. }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```