C Programming Notes

- #. C Programming Language
  - A . History of C.
  - B. Advantages of C
  - 2. A Basic C Program
- 3. Basic Structure of C Program :
- 4. Executing a C Program
- 5.Constants :
  - Integer constant :
    - A sequence of digits is referred as integer constant. There are three types of integer constants.
    - 1. Decimal integer constant :
    - 2. Octal integer constant :
    - 3. Hexadecimal integer constant:
  - Real constant :
  - Character constant :
  - String constant:
  - Backslash Character constant :
- 6.Variables :
  - How to Declare a Variable !
    - DECLARING & INITIALIZING C VARIABLE:
    - 1. EXAMPLE PROGRAM FOR LOCAL VARIABLE IN C:
      - OUTPUT:
    - 2. EXAMPLE PROGRAM FOR GLOBAL VARIABLE IN C:
      - OUTPUT:
- 6. Data Types in C Programming.
  - Data types determine the type of data a variable will hold.
- 7. Operators in C :
  - 1. Arithmetic Operators :
  - 2. Relational Operators :
  - 3. Logical Operators :
  - 4. Bitwise Operators :
  - 5. Assignment Operators :
  - 6. Special Operators :
  - 7. Conditional Operators :
- 8. Managing Input and Output Operations :
  - Basically there are some functions for input/output of data, they are:
- 9. Decision Making and Branching in C Programming :
  - 1.If statement :
    - The decision-making expression 'If' is the most powerful in the C language. As a result, it's used to control when statements are executed. The word 'if' is used in combination with an expression. It's a branching or decision-making statement with two options.
    - A. Simple if statement :
    - B. If else if statement:
    - C. Nested if statement :
    - D. The if… else… if ladder:
  - 2. The switch statement :
  - 3. Conditional operator statement :
  - 4. goto statement :
- 10. Decision Making and Looping in C Programming:
  - 1. The while statement :
  - 2. The do while statement :
  - 3. The for statement :
- Functions:
  - Predefined Functions:

#. C Programming Language

A . History of C.

- C is a general-purpose programming language **created** by **Dennis Ritchie** at the Bell Laboratories in 1972. It is a very popular language, despite being old. C is strongly associated with UNIX, as it was developed to write the UNIX operating system.
- C is the general programming language.
- C language stems from the BCPL language (Basic Combined Programming Language) and **C is developed** by **Dennis Ritchie** (a British computer scientist). C is preceding indirectly through B language via BCPL. **B language was introduced** by **Ken Thompson** (an American computer designer).
- BCPL and B are type-less languages (A type-less language would mean it allocates the same amount of memory for all kinds of data).

B. Advantages of C

1. **Almost all popular languages are built on top of the C language:**

   For example, All of these languages are C-level languages, such as Python, Java, JavaScript, Rubi, and C Shark. It simply implies that their libraries are built-in C code, which is then translated into a higher-level language such as JavaScript or java.

2. **It is the Coding language of choice for Kernel development:**

   The kernel is the central part of the operating system. It manages operations of the computer and the hardware most (notably) memory and CPU time. Because of this, C is our first pick when we need anything quick on our system.

3. **Easy to write:**

Another reason why C is so popular as an efficient language among programmers is that it allows them to create their software without having to worry about syntax errors. If you're not familiar with coding, use yourself to create more efficient and effective solutions than those created by other programming languages.

4. **Easy to understand :**

   One of the main reasons why people prefer C to other programming languages is its simplicity. C is a particularly portable language since it allows you to write programs that are faster and more efficient. As a result, C is simpler to learn than any other language. The concepts of C are straightforward to understand because there aren't many keywords or symbols.

5. **Presence of many Libraries :**

   C language provides many built-in functions consisting of system-generated functions and user-defined functions.

2. A Basic C Program

- To begin, the "HelloWorld program is the first and most basic c program you will learn in any programming language. All that is required is to display the word "Hello World" on the computer screen. Let's have a look at the software and see if we can figure out what the terms mean.
- To begin, the "Hello World program is the first hand most basic c program you will learn in any programming language. All that is required is to display the word "Hello World" on the computer screen. Let's have a look at the software and see if we can figure out what the terms mean.

```
// Simple C program to display "Hello World"

// Header file for input output functions
#include <stdio.h>

// main function -
// where the execution of program begins

int main()
{
// prints hello world
    printf("Hello World");
 return 0;
}
```

1. `# include <stdio.h>`

   Directives are all lines in C that begin with the pound (#) sign. The compiler calls a preprocessor program to process these statements. The #include directive instructs the compiler to include a file, and #includestdio. h> instructs the compiler to include the Standard Input Output header file, which contains definitions for all standard input/output library functions.

2. `int main( )`

   This line declares the "main" function, which returns data of the integer type. A function is a collection of statements that accomplish a specified activity. The main() function, no matter where it is in the program, is where the execution of every C program starts. As a result, every C program must include a main() function, which is where the program's execution begins.

3. `{ and }`

   The beginning of the main function is indicated by the opening braces '', and the ending of the main function is shown by the closing braces ''. The blocks are everything in between these two that make up the primary function's body.

4. `printf("hello world");`

   The compiler is instructed to display the message "Hello World" on the screen by this line. In C, this is referred to as a statement. Every remark is intended to accomplish a specific goal. To finish a statement, use the semicolon ';'. The semicolon character at the conclusion of the sentence denotes that the statement is coming to a close. On the stdout console, the printf() function is used to print a character stream of data. The output device sees everything contained in " ".

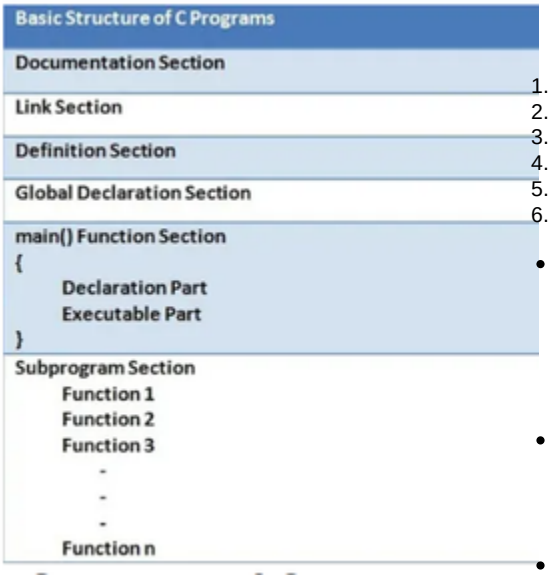
5. `return 0 ;`

This is also a declaration. This statement is used to signify the end of a function and to return a value from it. This statement is used in functions to return the results of the actions that the function has executed.

- **Indentation** :

  As you can see the printf and the return statement have been indented or moved to the right side. This is done to make the code more readable. In a program as Hello World, it does not seem to hold much relevance but as the program becomes more complex, it makes the code more readable and less error-prone. Therefore, one must always use indentations and comments to make the code more readable.

3. Basic Structure of C Program :



Any C program is consists of 6 main sections.

1. ***Documentation Section***
2. ***Link Section***
3. ***Definition Section***
4. ***Global Declaration Section***
5. ***Main( ) Function Section***
6. ***Subprogram Section***

- `Documentation Section`

  This section contains remark lines that provide the programmer's name, the author's name, and other information such as the time and date the programme was written. The **documentation part assists anyone in getting a general understanding of the program.**

- `Link Section`

  The header files for the functions used in the application are found in the link section. It **tells the compiler how to connect functions from the system library together.**

- `Definition Section`

  The defining section **contains all of the symbolic constants**. Symbolic constants are what macros are called.

- `Global Declaration Section`

  Global variables are declared in the global declaration section and can be used elsewhere in the programme. The user-defined functions are also declared in this section.

- `Main( ) Function Section`

  Every C programme must have at least one main() function section. This section is divided into two sections: declaration and executable. All variables used in the executable segment are declared in the declaration section. Between the opening and closing braces, these two portions must be written. A semicolon must be used at the end of each statement in the **declaration and executable sections (;)**. The **pro gram's execution begins** with the opening of braces and finishes with the closure of braces.

- `Subprogram Section`

  All of the **user-defined functions** that are **utilized to complete a certain task** are found in the subprogram section. The main() method calls these user-defined functions.

4. Executing a C Program

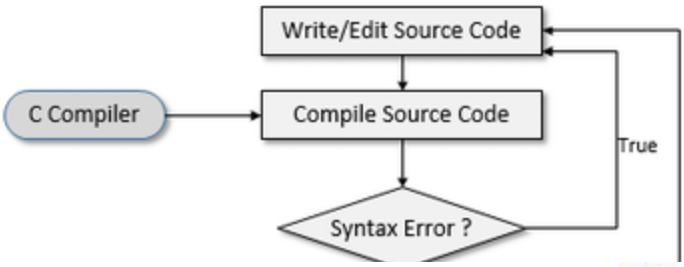
> **Process of compiling & Running C Program**

> Compilation and execution are important in the C programming language.

> The first is that what we write as programmers is in a high-level language (C, C++, Java), which the computer can not understand.

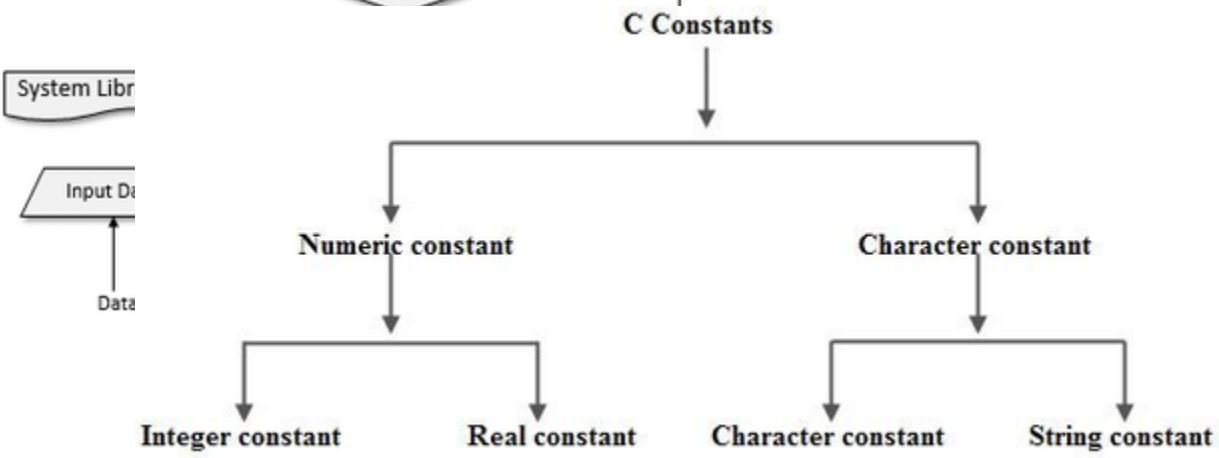
> The only language that the computer knows is Binary, which is a language made up of 1s and 0s.

1. Use any standard editor, such as Code blocks, to create a C-Program source file.
2. Use the C compiler to compile the source file.
3. Change the source code of the program. Otherwise, it will generate an object code after successful compilation if any errors are identified.



4. During the linking process, system library files are linked.
5. After the linking phase, it will generate executable code.
6. During the execution of a programme, check for any logical or data errors. > If any logical errors are discovered, modify the source code and recompile the source file. > If there is a data error, run the application again with the correct data.
7. If everything is in order, it will provide the desired results.

5.Constants :



- **Constants are data values that remain the same each time a programme is run**. The constants are unlikely to change. Values are fixed into the source code as real constants. **Constants in C include** **Numeric Constants and Character Constants**, among others. numerical constants are divided into integer constants and real constants. Similarly character constants are divided into string and character constants.

Integer constant :

A sequence of digits is referred as integer constant. There are three types of integer constants.

1. Decimal integer constant :

- Decimal integer constant consists of **digits from 0 to 9** preceding by the optional minus or plus sign.
- **E.x. 321, 123, -43 , 0 ,8349, etc.**

2. Octal integer constant :

- Octal integer constant consists of any combination of **digits from 0 to 7** with the **leading of 0** (Zero).
- **E.x. 036, 02734, 08, 026286, etc.**

3. Hexadecimal integer constant:

- The **0 digit** is followed by either an **x or an X**, then any combination of the **digits 0 through 9** and the characters a through **f or A** through F. The numbers 10 through 15 are represented by the letters A (or a) through F (or f).
- **E.x. 0x90, 0X640, 0x8, etc.**

Real constant :

- Constants are constant values in C that do not change throughout programme execution. A real constant is made up of a whole number, a decimal point, and the fractional part of the value.
- **Ex. 0.0083, -0.75 .**

Character constant :

- Enclosing a single character from the representable character set within single quote marks (' ') creates a "character constant."
- **Ex. ' 7 ', ' x ' , ' : '**

Char and string constant value

- Char

```
char c;
c = 'A'; // d = 65;
```

- String

```
printf("string is array of char!!!");
printf("example of escape sequence is \n");
```

String constant:

- String constant is a sequence of characters enclosed in Double Quotes. (" "). It can be any combination of letters, numbers, or special characters, and it must be enclosed in double quotations.

  Ex. " Hello " , "2876abc ", " ss12; "

Backslash Character constant :

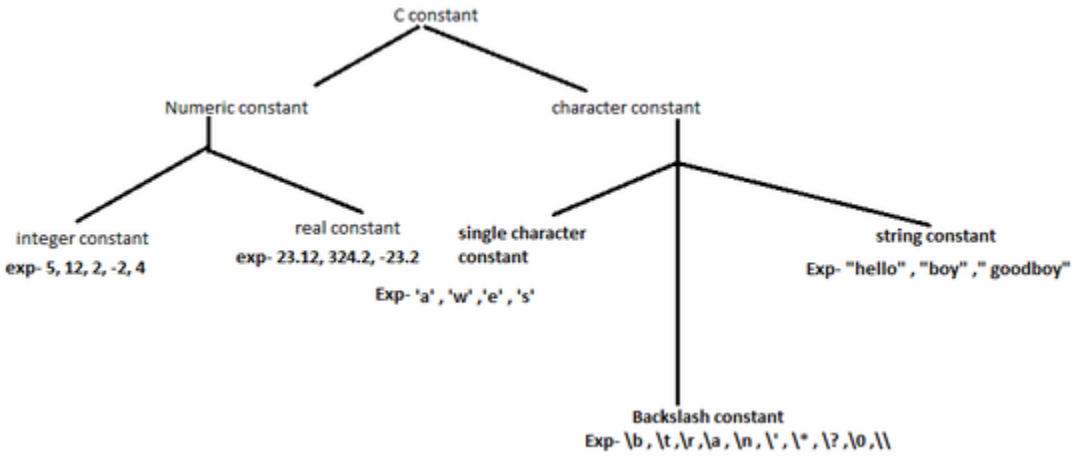
- **C supports some special backslash constants that are used in Output Functions.**

  E.x.

  \a : audible alert

  \b : back space

  \n : new line, etc.



6.Variables :

- Variable is a data name that can be used to store a Data value.
- **Unlike constants that remains unchanged during the execution of a program.**
- **A Variable may take different values at different times of execution.**
- **A Variable name can be chosen by a programmer in a meaningful way so as to reflect its function or nature in the program.**
- Examples : **Average , Height, Total, class_strength, counter_1, etc**

- **Variable names may consist of Letters, Digits and the underscore character, subject to the following conditions:**
- **They must begin with a Letter. Some systems permits the use of underscore character.**
- **ANSI standard recognizes a length of 31 characters. However length should not be more than 8 characters, Since only first 8 characters are treated as significant by many compilers.**
- **Uppercase and Lowercase are significant. E.x. Total, total and Total are 3 different variables.**
- **it should not be a Keyword. Ex. for, if, printf, etc. This can't be used as Variable name.**
- **White Spaces are not allowed.**
- **Examples of valid Variables** : **John, Value, Delhi, x1, sum1, etc.**

How to Declare a Variable !

DECLARING & INITIALIZING C VARIABLE:

- Variables should be declared in the C program before to use.
- Memory space is not allocated for a variable while declaration. It happens only on variable definition.
- Variable initialization means assigning a value to the variable.

Type	Syntax
Variable declaration	data_type variable_name; Example: int x, y, z; char flat, ch;
Variable initialization	data_type variable_name = value; Example: int x = 50, y = 30; char flag = 'x', ch='l';

- **THERE ARE THREE TYPES OF VARIABLES IN C PROGRAM THEY ARE,**

  Local variable

  Global variable

  Environment variable

1. EXAMPLE PROGRAM FOR LOCAL VARIABLE IN C:

- The scope of local variables will be within the function only.
- These variables are declared within the function and can't be accessed outside the function.
- In the below example, m and n variables are having scope within the main function only. These are not visible to test function.
- Like wise, a and b variables are having scope within the test function only. These are not visible to main function.

```
#include<stdio.h>
void test();
int main()
{
   int m = 22, n = 44;
   printf("\n values : m = %d and n = %d", m, n);
   test();

       // m, n are variables of main function
       //*m and n variables are having scope within this main function
only.
       //These are not visible to test funtion.
       /* If you try to access a and b in this function,
       you will get 'a' undeclared and 'b' undeclared error */

  }

void test()
```

```
{
    int a = 50, b = 80;
    printf("\n values : a = %d and b = %d", a, b);}

       // a, b are variables of test function
        /*a and b variables are having scope within this test function
only.
        These are not visible to main function.*/
        /* If you try to access m and n in this function,you will get
'm' undeclared and 'n' undeclared
        error */
```

OUTPUT:

2. EXAMPLE PROGRAM FOR GLOBAL VARIABLE IN C:

- The scope of global variables will be throughout the program. These variables can be accessed from anywhere in the program.
- This variable is defined outside the main function. So that, this variable is visible to main function and all other sub functions.

```
#include<stdio.h>
void test();
int m = 22, n = 44;
int a = 50, b = 80;
int main()
{
    printf("All variables are accessed from main function");
    printf("\nvalues: m=%d:n=%d:a=%d:b=%d", m,n,a,b);
    test();
}

    void test()
  {

  printf("\n\nAll variables are accessed from" \   " test function");
  printf("\nvalues: m=%d:n=%d:a=%d:b=%d", m,n,a,b);
   }
```

OUTPUT:

```
All variables are accessed from main function
values : m = 22 : n = 44 : a = 50 : b = 80
All variables are accessed from test function
values : m = 22 : n = 44 : a = 50 : b = 80
```

6. Data Types in C Programming.

Data types determine the type of data a variable will hold.

- **The C programming language provides a predefined set of data types for dealing with different forms of data that we can use in our programme. These data types have different storage capacities.**
- **C Supports 2 types of Data types :**

    Primary Data Types

    Derived Data types

Following are the examples of some very common data types used in C:

- **Integer Type :** Integers are used to store a whole number.
- **Floating Point Type :** It's used to store decimal integers (numbers with floating point values). Floating types are used to store real numbers.
- **Character Type :** Character types are used to store character value.
- **Void Type :** Void type means no value. This is usually used to specify the type of functions which returns nothing.

7. Operators in C :

An operator is simply a symbol that is used to perform operations.

- **C supports** a rich set of built-in operators.
- **Operator** is a symbol that tells the computer to perform some certain mathematical or logical operations.
- **Operators** can be used in a program to manipulate the data and variables.

Operators in C Language :

Arithmetic operators

Relational operators

Logical operators

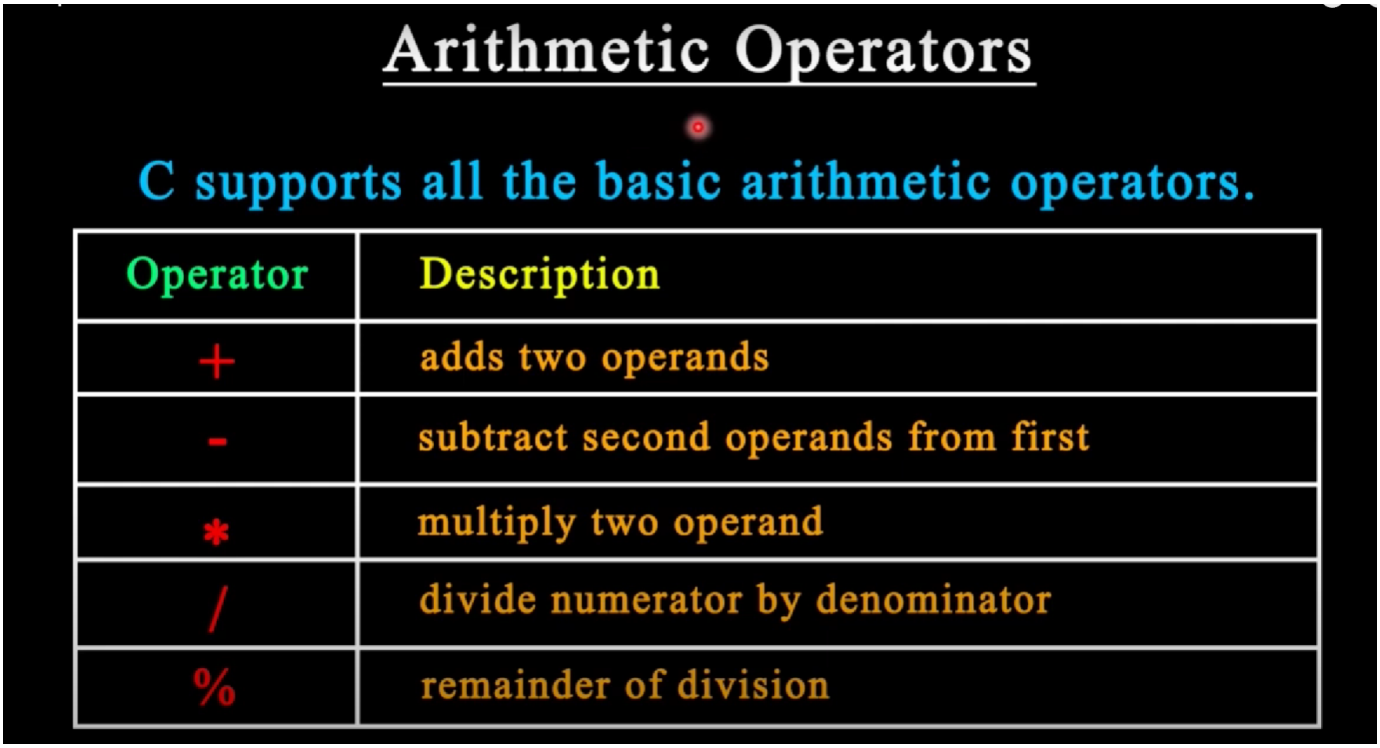
Bitwise operators

Assignment operators

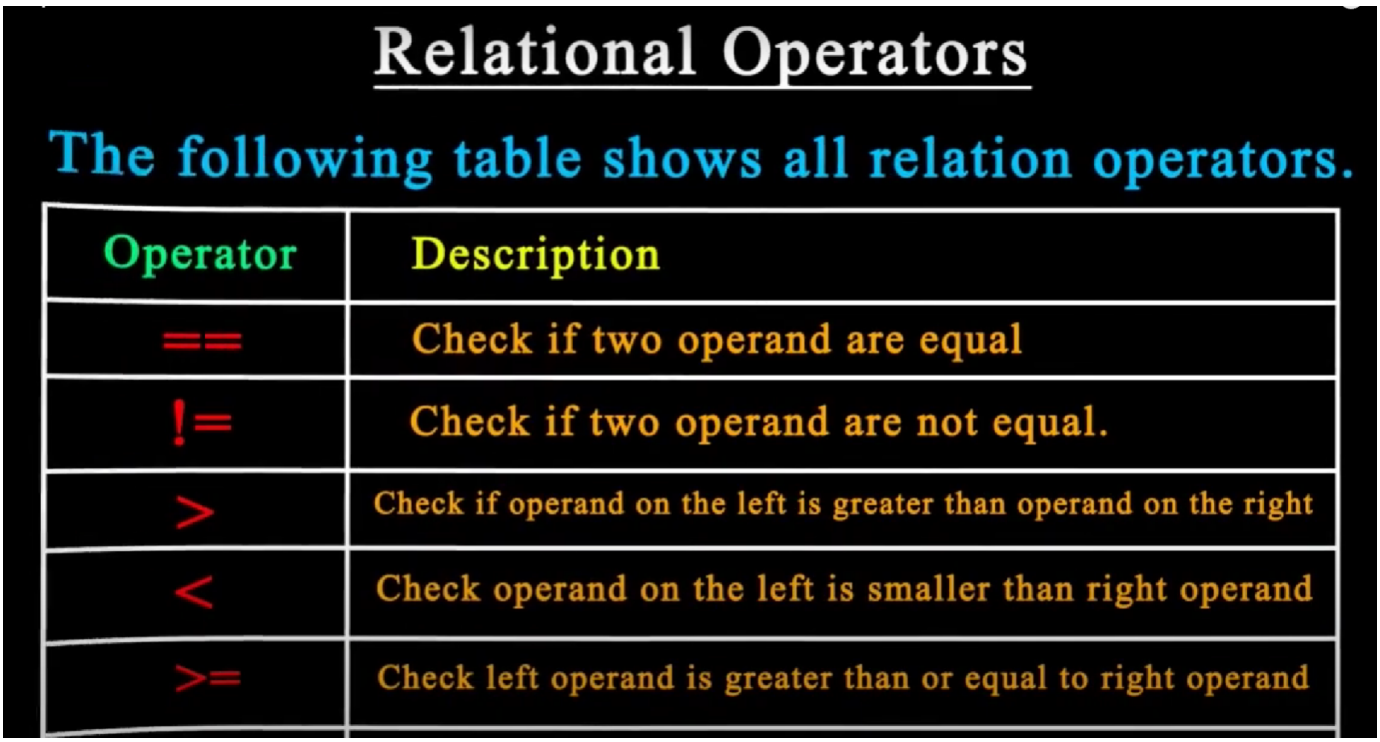
Conditional Operators

Special operators

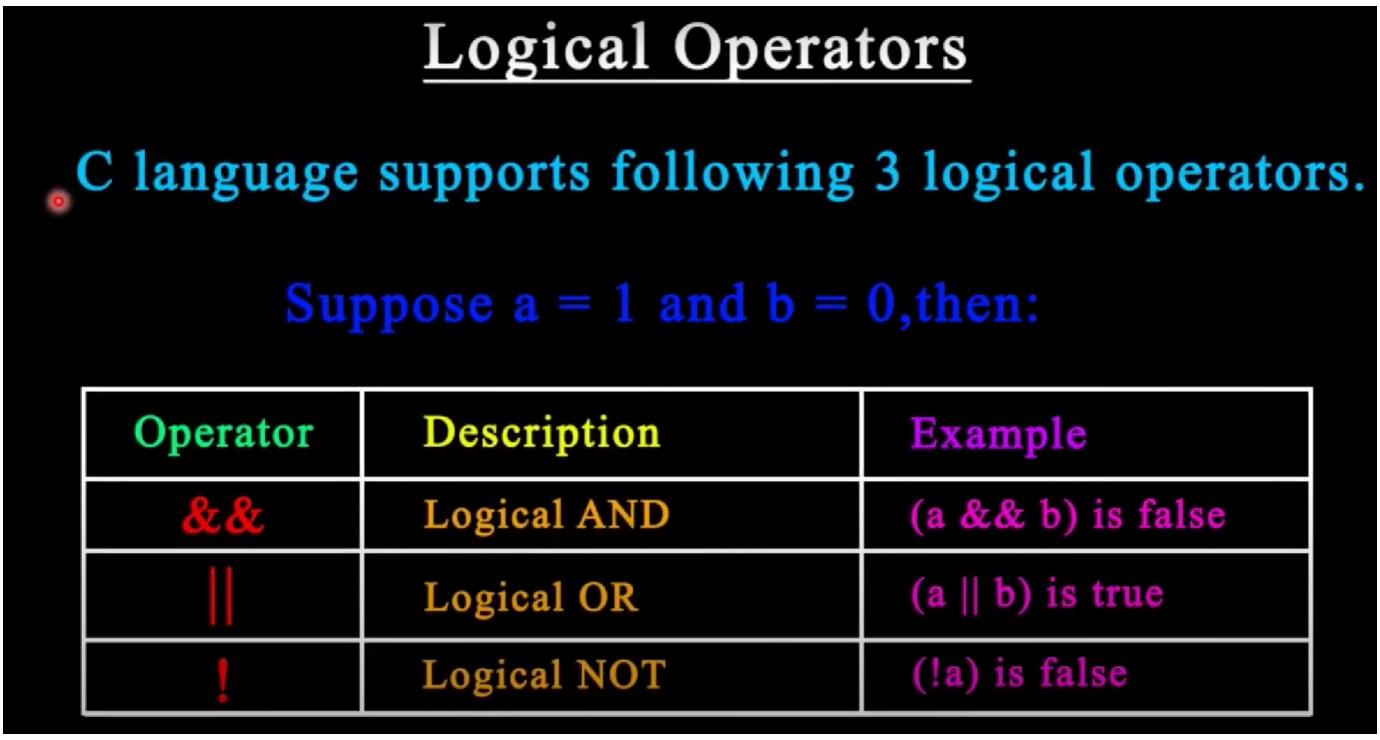
1. Arithmetic Operators :



2. Relational Operators :



3. Logical Operators :



4. Bitwise Operators :

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

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

5. Assignment Operators :

Assignment operators supported by C language are as follows.

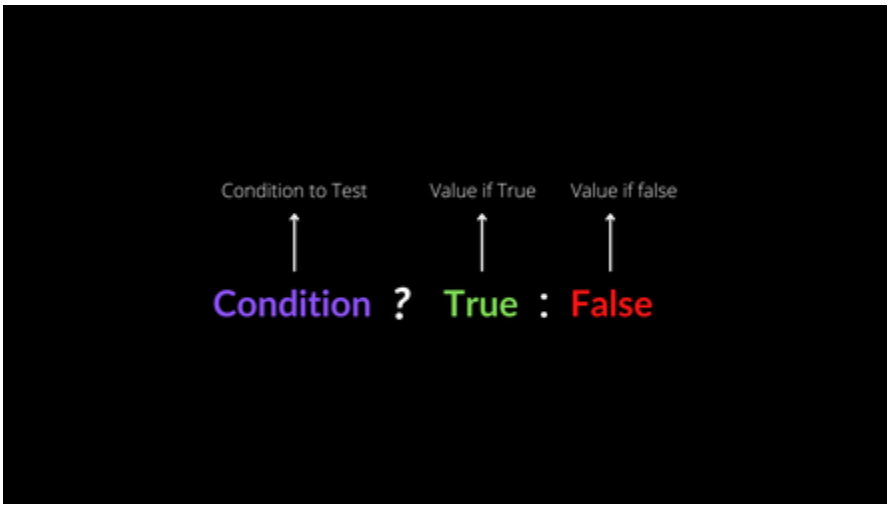
Operator	Description	Example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	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

6. Special Operators :

C language supports following 3 Special operators.

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

7. Conditional Operators :



8. Managing Input and Output Operations :

- **Reading, Processing and Writing Data are the three essential functions of a computer program. So, most programs take some data as input as display the process data often known as information or Results on a suitable medium.**
- **In C, how do you manage input and output operations?**
  - Standard input, or stdin, is a data that accepts input from devices like as the keyboard. Standard output, often known as stdout, is used to send data to a device like a monitor. Programmers must include the stdio.h header-file in their programmers in order to use I/O functions.
- **All Input/ Output operations carried out through functions calls such as printf and scanf.**
- **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.
- The C programming language has many built-in functions for reading any given input and displaying data on the screen when the result is needed.
- `#include<stdio.h>` **is a command to search and include the contents of the header file stdio.h in the program.**

Basically there are some functions for input/output of data, they are:

Scanf( )

printf( )

getchar( )

putchar( )

9. Decision Making and Branching in C Programming :

- **It supports sequential programme statements in C programming, which execute one statement after another.**
- **C language possesses decision making and branching capabilities by supporting the following statements:**

    1. If statement.

    2. Switch statement.

    3. Conditional operator statement.

    4. goto statement.

1.If statement :

The decision-making expression 'If' is the most powerful in the C language. As a result, it's used to control when statements are executed. The word 'if' is used in combination with an expression. It's a branching or decision-making statement with two options.

Syntax :

```
if(test expression){
    Body of if;
}
```

Depending upon the **complexity and conditions** to be tested if can be further subdivided.

A. Simple if

B. if.....else

C. nested if...else

D. if...else...if ladder

A. Simple if statement :

- **If statement is responsible for modifying the flow of execution of a program.**

Syntax:

```
if(test expression)
{
        Body of if;
}
```

- The 'if' body can be a single statement or a group of statements. The if body is run when the test expression returns a true value. If the condition is false, control is transferred to statement X. In both circumstances, the statement X will be executed.

B. If else if statement:

An extension to simple if is called if.. else statement.

Syntax:

```
if(test expression){
       Body of if;
}
```

```
else
{
      Body of else;
}
```

- The condition is first tested in a simple if statement. If true, the body of the if statement is executed, then the next section is skipped, and the control is passed to the statement x. We've added another block called else, which contains a series of lines to execute when the condition is false. When the condition fails, the flow moves to the else part after skipping the body of the if condition. The condition may return either 'true' or 'false,' in which case the statement x is executed in both situations.

C. Nested if statement :

- It is used when a series of decisions are involved and have to use more than one if…else statement.

Syntax :

```
   if(condition 1)   {
            if(condition 2)
            {
                body of if;
            }
            else
            {
               Body of else;
            }
   }
   else
   {
        Body of else;
   }
```

D. The if… else… if ladder:

- To take multipath decisions or chain of ifs then we use the if …else…if ladder. It is in the following general form

Syntax :

```
if(condition 1)
   statement 1;
   else if(condition 2)
       statement 2;
       else if(condition 3)
          statement 3;
          else
             statement 4;
```

statement x;

The conditions are evaluated from the top of the ladder down words as soon as a true condition is found. The compiler executes the ladder till a true condition is found, and when found a true condition execute the statements associated with it. After executing the statements the control passes to the statement x.

2. The switch statement :

- **If is a conditional statement, and switch is a selection statement, meaning that we use 'switch' to choose one of many options. The selective statement 'switch' is a multi-way decision statement, meaning that we can direct control flow to many paths using a single expression. When a match is found, the 'Switch' statement compares the value of a specified expression or variable to a list of case values, and the block of instructions associated with that case is executed.**
- **Syntax :**

```
switch(expression) {
     case constant1 :
                              block1;
                              break;
     case constant2 :
                              block2;
                              break;
     case constant n:
                              block n;
                              break;
                                  .
                                  .
     default:
                default block;
                 break;
}
```

- The value of the expression is correctly checked against the case's constants when the 'switch' is executed. When a case constant is located, the block of codes associated with it is executed, and if no match is found until the last case constant, the default case block is executed. An optional case is the default. Control flows to the statement x if it is not present and no matches are found.

3. Conditional operator statement :

- **The `conditional operator` is similar to the if-else statement in that it performs the same algorithm as the if-else statement, but it looks smaller and helps in the shortest possible writing of if-else statements.**
- **Syntax :**

```
variable = Expression1 ? Expression2 : Expression3
```

- **Conditional Operator :** takes three operands to work, hence they are also called ternary operators.
  - **Working:**
    - **Expression1 is the condition to be checked in this case. Expression2 will be executed and the result will be returned if the condition(Expression1) is True. If condition(Expression1) is false, Expression3 will be run and the result returned.**

4. goto statement :

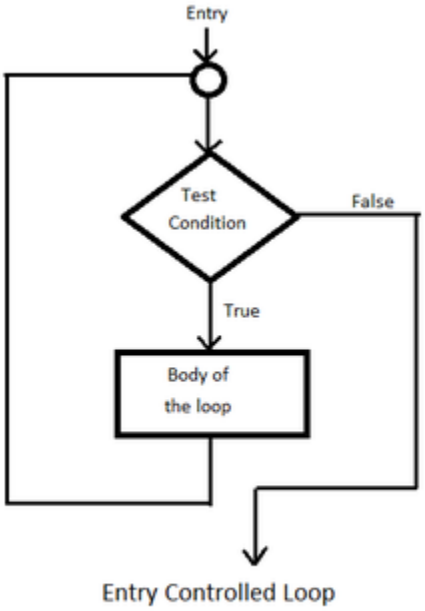
- **The goto statement is a type of jump statement that is also known as an unconditional jump statement. Within a function, the goto statement can be used to jump from one place to another.**
- **Syntax :**

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

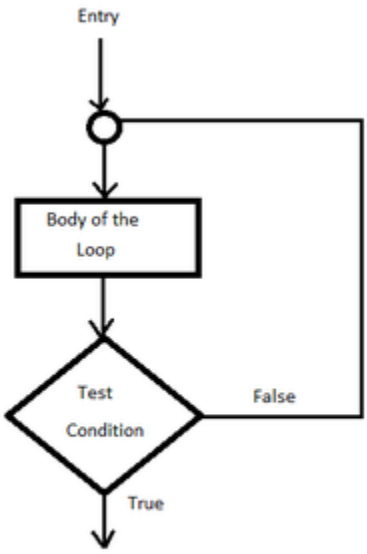
- **Label is an identifier in this syntax. When the program's control hits the goto statement, the control jumps to the label: and executes the code behind it.**

10. Decision Making and Looping in C Programming:

- Decision statements are used to check for specific conditions before proceeding.
- If the condition in the decision statement is true, one set of instructions is executed; if the condition is false, other set of instructions is executed.
- So, depending on position of control statement in the loop, a control structure is either classified as **Entry-controlled loop or Exit-controlled loop**



Entry Controlled Loop



Exit Controlled Loop

A looping Process, in general, would include the following four steps:

- **Setting and initialization of condition variable.**
- **Execution of the statements in the loop.**
- **Test for a specified value off the condition variable for execution of the loop.**
- **Incrementing or updating the condition variable.**

The C language, there are three constructs for performing looping operations.

- **The while statement**
- **The do while statement**
- **The for statement**

1. The while statement :

- In C programming, the while loop is the most fundamental loop. The while loop has only single control condition and runs as long as it is true. An entry-controlled loop is one in which the loop's condition is tested before the loop's body is executed.

Syntax:

```
While (condition)
{
   statement(s);
   Incrementation;
}
```

2. The do while statement :

- The do-while statement **lets you repeat a statement or compound statement until a specified expression becomes false**.

Syntax:

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

3. The for statement :

- The for statement allows users to specify how many times a statement or compound statement should be repeated. A for statement's body is executed one or more times until an optional condition is true.

Syntax:

```
for (statement 1; statement 2; statement 3)
{
// code block to be executed
}
```

Note : **The remaining notes, which are from functions and algorithms, will be updated soon.**

Functions:

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Predefined Functions:

- So it turns out you already know what a function is. You have been using it the whole time.
- For example, main() is a function that executes code, and printf() is a function that outputs/prints text to the screen:

```
• int main() {
    printf("Hello World!");
```

```
  return 0;
}
```

Create a Function:

- To create (often referred to as *declare*) your own function, specify the name of the function, followed by parentheses `()` and curly brackets `{}`:
- **Syntax:**

```
void myFunction() {
  // code to be executed
}
```

Example Explained

- `myFunction()` is the name of the function
- `void` means that the function does not have a return value.
- Inside the function (the body), add code that defines what the function should do

Call a Function:

- Declared functions are not immediately executed.
- They are "saved for later use" and will be executed when called upon.
- To invoke a function, type its name followed by two parentheses () and a semicolon;
- When myFunction() is called in the following example, it prints a text (the action):

```
 // Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

- A function can be called multiple times:

```
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction();
  myFunction();
```

```
  myFunction();
  return 0;
}

// I just got executed!
// I just got executed!
// I just got executed!
```

C Function Parameters:

Parameters and Arguments:

- As a parameter, information can be passed to functions. Within the function, parameters function as variables.
- Parameters are specified inside the parentheses after the function name. You can enter as many parameters as you want, separated by a comma:
- **Syntax:**

```
returnType functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

- **Example:**

```
void myFunction(char name[]) {
  printf("Hello %s\n", name);
}

int main() {
  myFunction("Liam");
  myFunction("Jenny");
  myFunction("Anja");
  return 0;
}
```

- Output:

```
Hello Liam
Hello Jenny
Hello Anja
```

Multiple Parameters

- Inside the function, you can add as many parameters as you want:
- Example:

```
```

```
void myFunction(char name[], int age) {
  printf("Hello %s. You are %d years old.\n", name, age);
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}
```

- Output:

```
Hello Liam. You are 3 years old.
Hello Jenny. You are 14 years old.
Hello Anja. You are 30 years old.
```

Return Values:

- The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as `int` or `float`, etc.) instead of `void`, and use the `return` keyword inside the function:

```
int myFunction(int x) {
  return 5 + x;
}

int main() {
  printf("Result is: %d", myFunction(3));

  return 0;
}

// Outputs 8 (5 + 3)
```

- This example returns the sum of a function with **two parameters**:

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  printf("Result is: %d", myFunction(5, 3));

  return 0;
```

```
}

// Outputs 8 (5 + 3)
```

- You can also store the result in a variable:

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  int result = myFunction(5, 3);
  printf("Result is = %d", result);

  return 0;
}
// Outputs 8 (5 + 3)
```

C Function Declaration and Definition

Function Declaration and Definition

A function consist of two parts:

- **Declaration:** the function's name, return type, and parameters (if any)
- **Definition:** the body of the function (code to be executed)
- Syntax:

```
void myFunction() { // declaration
  // the body of the function (definition)
}
```

- For code optimization, it is recommended to separate the declaration and the definition of the function.

  You will often see C programs that have function declaration above `main()`, and function definition below `main()`. This will make the code better organized and easier to read:
- Example:

```
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
```

```
void myFunction() {
  printf("I just got executed!");
}
```

C Recursion

- Recursion is a process that occurs when a function calls a copy of itself to work on a smaller problem.
- Any function that calls itself is referred to as a recursive function, and such function calls are referred to as recursive calls.
- Recursion entails a large number of recursive calls.
- However, it is critical to impose a recursion termination condition.
- Although recursion code is shorter than iterative code, it is more difficult to understand.
- Recursion cannot be used for every problem, but it is most useful for tasks that can be defined in terms of similar subtasks.
- Recursion can be used to solve sorting, searching, and traversal problems, for example.
- Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.
- In the following example, recursion is used to calculate the factorial of a number.

```
#include <stdio.h>
int fact (int);
int main()
{
    int n,f;
    printf("Enter the number whose factorial you want to calculate?");
    scanf("%d",&n);
    f = fact(n);
    printf("factorial = %d",f);
}
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if ( n == 1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
```

- **Output:**

```
Enter the number whose factorial you want to calculate?5
factorial = 120
```

Recursive Function

- A recursive function performs tasks by breaking them down into subtasks.
- The function defines a termination condition that is satisfied by a specific subtask.
- The recursion is then terminated, and the function returns the final result.
- The case in which the function does not recur is known as the base case, whereas the case in which the function repeatedly calls itself to perform a subtask is known as the recursive case.
- This format can be used to write all recursive functions.
- The following pseudocode is for writing any recursive function.

```
if (test_for_base)
{
    return some_value;
}
else if (test_for_another_base)
{
    return some_another_value;
}
else
{
    // Statements;
    recursive call;
}
```

Example of recursion in C

- Let's see an example to find the nth term of the Fibonacci series.

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
    int n,f;
    printf("Enter the value of n?");
    scanf("%d",&n);
    f = fibonacci(n);
    printf("%d",f);
}
int fibonacci (int n)
{
    if (n==0)
    {
    return 0;
    }
    else if (n == 1)
    {
        return 1;
    }
    else
```

```
    {
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

- **Output:**

```
Enter the value of n?12
144
```

Pointers

- A pointer is a variable whose value is the address of another variable, i.e., the memory location's direct address.
- A pointer, like any variable or constant, must be declared before it can be used to store any variable address.
- A pointer variable declaration takes the following general form:

```
type *var-name;
```

- Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable.
  - The asterisk * used to declare a pointer is the same asterisk used for multiplication.
  - However, in this statement the asterisk is being used to designate a variable as a pointer.
  - Take a look at some of the valid pointer declarations

```
• int    *ip;    /* pointer to an integer */
  double *dp;    /* pointer to a double */
  float  *fp;    /* pointer to a float */
  char   *ch     /* pointer to a character */
```

- The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

- There are a few important operations, which we will do with the help of pointers very frequently.

  (a) We define a pointer variable,

  (b) assign the address of a variable to a pointer and

  (c) finally access the value at the address available in the pointer variable.

  This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations

Example:

```
#include <stdio.h>

int main () {

   int  var = 20;   /* actual variable declaration */
```

```
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

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

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

- It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program

```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

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

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

Structures in C

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name.

Define Structures

Before you can create structure variables, you need to define its data type. To define a struct, the `struct` keyword is used.

Syntax of struct

```
struct structureName {
  dataType member1;
  dataType member2;
  ...
};
```

For example,

```
struct Person {
  char name[50];
  int citNo;
  float salary;
};
```

Here, a derived type `struct Person` is defined. Now, you can create variables of this type.

Create struct Variables

- When a `struct` type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.
- Here's how we create structure variables:

```
struct Person {
  // code
};

int main() {
  struct Person person1, person2, p[20];
  return 0;
}
```

Another way of creating a `struct` variable is:

```
struct Person {
   // code
} person1, person2, p[20];
```

In both cases,

- person1 and person2 are `struct Person` variables
- p[] is a `struct Person` array of size 20.

Access Members of a Structure

There are two types of operators used for accessing members of a structure.

1. `.` - Member operator
2. `->` - Structure pointer operator (will be discussed in the next tutorial)

Suppose, you want to access the salary of person2. Here's how you can do it.

```
person2.salary
```

- Example 1 on structures in c

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

// create struct with person1 variable
struct Person {
  char name[50];
  int citNo;
  float salary;
} person1;

int main() {

  // assign value to name of person1
  strcpy(person1.name, "George Orwell");

  // assign values to other person1 variables
  person1.citNo = 1984;
  person1. salary = 2500;

  // print struct variables
  printf("Name: %s\n", person1.name);
  printf("Citizenship No.: %d\n", person1.citNo);
  printf("Salary: %.2f", person1.salary);

  return 0;
}
```

Output

```
```

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

In this program, we have created a `struct` named `Person`. We have also created a variable of `Person` named person1.

In `main()`, we have assigned values to the variables defined in `Person` for the person1 object.

```
strcpy(person1.name, "George Orwell");
person1.citNo = 1984;
person1. salary = 2500;
```

Notice that we have used `strcpy()` function to assign the value to person1.name

This is because name is a `char` array (C-String) and we cannot use the assignment operator = with it after we have declared the string.

Keyword typedef

We use the `typedef` keyword to create an alias name for data types. It is commonly used with structures to simplify the syntax of declaring variables.

For example, let us look at the following code:

```
struct Distance{
  int feet;
  float inch;
};

int main() {
  struct Distance d1, d2;
}
```

We can use `typedef` to write an equivalent code with a simplified syntax:

```
typedef struct Distance {
  int feet;
  float inch;
} distances;

int main() {
  distances d1, d2;
}
```

Example 2: C typedef

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

// struct with typedef person
typedef struct Person {
  char name[50];
  int citNo;
  float salary;
} person;

int main() {

  // create  Person variable
  person p1;

  // assign value to name of p1
  strcpy(p1.name, "George Orwell");

  // assign values to other p1 variables
  p1.citNo = 1984;
  p1. salary = 2500;

  // print struct variables
  printf("Name: %s\n", p1.name);
  printf("Citizenship No.: %d\n", p1.citNo);
  printf("Salary: %.2f", p1.salary);

  return 0;
}
```

Output

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

Here, we have used `typedef` with the `Person` structure to create an alias `person`.

```
// struct with typedef person
typedef struct Person {
  // code
} person;
```

Now, we can simply declare a `Person` variable using the `person` alias:

```
// equivalent to struct Person p1
person p1;
```

Nested Structures

You can create structures within a structure in C programming. For example,

```
struct complex {
  int imag;
  float real;
};

struct number {
  struct complex comp;
  int integers;
} num1, num2;
```

Suppose, you want to set imag of num2 variable to **11**. Here's how you can do it:

```
num2.comp.imag = 11;
```

Example 3: C Nested Structures

```
#include <stdio.h>

struct complex {
  int imag;
  float real;
};

struct number {
  struct complex comp;
  int integer;
} num1;

int main() {

  // initialize complex variables
  num1.comp.imag = 11;
  num1.comp.real = 5.25;

  // initialize number variable
  num1.integer = 6;
```

```
  // print struct variables
  printf("Imaginary Part: %d\n", num1.comp.imag);
  printf("Real Part: %.2f\n", num1.comp.real);
  printf("Integer: %d", num1.integer);

  return 0;
}
```

Output

```
Imaginary Part: 11
Real Part: 5.25
Integer: 6
```

Why structs in C?

Suppose you want to store information about a person: his/her name, citizenship number, and salary. You can create different variables name, citNo and salary to store this information.

What if you need to store information of more than one person? Now, you need to create different variables for each information per person: `name1`, `citNo1`, `salary1`, `name2`, `citNo2`, `salary2`, etc.

A better approach would be to have a collection of all related information under a single name `Person` structure and use it for every person.

C Unions

- A union is a user-defined type similar to struct in C except for one key difference.
- Structures allocate enough space to store all their members, whereas **unions can only hold one member value at a time**.

How to define a union?

We use the `union` keyword to define unions. Here's an example:

```
union car
{
  char name[50];
  int price;
};
```

The above code defines a derived type `union car`.

Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

```
union car
{
  char name[50];
  int price;
};

int main()
{
  union car car1, car2, *car3;
  return 0;
}
```

Another way of creating union variables is:

```
union car
{
  char name[50];
  int price;
} car1, car2, *car3;
```

In both cases, union variables car1, car2, and a union pointer car3 of `union car` type are created.

Access members of a union

We use the `.` operator to access members of a union. And to access pointer variables, we use the `->` operator.

In the above example,

- To access price for `car1`, `car1.price` is used.
- To access price using `car3`, either `(*car3).price` or `car3->price` can be used.

Difference between unions and structures

Let's take an example to demonstrate the difference between unions and structures:

```
#include <stdio.h>
union unionJob
{
   //defining a union
   char name[32];
   float salary;
   int workerNo;
} uJob;

struct structJob
{
   char name[32];
   float salary;
```

```
   int workerNo;
} sJob;

int main()
{
   printf("size of union = %d bytes", sizeof(uJob));
   printf("\nsize of structure = %d bytes", sizeof(sJob));
   return 0;
}
```

Output

```
size of union = 32
size of structure = 40
```

Why this difference in the size of union and structure variables?

- Here, the size of sJob is 40 bytes because
  - **the size of** `name[32]` **is 32 bytes**
  - **the size of** `salary` **is 4 bytes**
  - **the size of** `workerNo` **is 4 bytes**
- However, the size of uJob is 32 bytes. It's because the size of a union variable will always be the size of its largest element. In the above example, the size of its largest element, (`name[32]`), is 32 bytes.
- With a union, all members share **the same memory**.
- Example: Accessing Union Members

```
#include <stdio.h>
union Job {
   float salary;
   int workerNo;
} j;

int main() {
   j.salary = 12.3;

   // when j.workerNo is assigned a value,
   // j.salary will no longer hold 12.3
   j.workerNo = 100;

   printf("Salary = %.1f\n", j.salary);
   printf("Number of workers = %d", j.workerNo);
   return 0;
}
```

Output

```
Salary = 0.0
Number of workers = 100
```

Structure	Union
We use the struct statement to define a structure.	We use the union keyword to define a union.
Every member is assigned a unique memory location.	All the data members share a memory location.
Change in the value of one data member does not affect other data members in the structure.	Change in the value of one data member affects the value of other data members.
You can initialize multiple members at a time.	You can initialize only the first member at once.
A structure can store multiple values of the different members.	A union stores one value at a time for all of its members
A structure's total size is the sum of the size of every data member.	A union's total size is the size of the largest data member.
Users can access or retrieve any member at a time.	You can access or retrieve only one member at a time.

Enum in C

Enumeration or Enum in C is a special kind of data type defined by the user. **It consists of constant integrals or integers that are given names by a user.** The use of enum in C to name the integer values makes the entire program easy to learn, understand, and maintain by the same or even different programmer.

- The keyword 'enum' is used to declare new enumeration types in C and C++. Following is an example of enum declaration.

```
// The name of enumeration is "flag" and the constant
// are the values of the flag. By default, the values
// of the constants are as follows:
// constant1 = 0, constant2 = 1, constant3 = 2 and
// so on.
enum flag{constant1, constant2, constant3, ....... };
```

- Variables of type enum can also be defined. They can be defined in two ways:

```
// In both of the below cases, "day" is
// defined as the variable of type week.

enum week{Mon, Tue, Wed};
enum week day;

// Or

enum week{Mon, Tue, Wed}day;
```

Example:

```
// An example program to demonstrate working
// of enum in C
#include<stdio.h>
```

```
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};

int main()
{
        enum week day;
        day = Wed;
        printf("%d",day);
        return 0;
}
```

Output:

```
2
```

- In the above example, we declared “day” as the variable and the value of “Wed” is allocated to day, which is 2. So as a result, 2 is printed.
- Another example of enumeration is:

```
// Another example program to demonstrate working
// of enum in C
#include<stdio.h>

enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,
          Aug, Sep, Oct, Nov, Dec};

int main()
{
   int i;
   for (i=Jan; i<=Dec; i++)
      printf("%d ", i);

   return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

- In this example, the for loop will run from i = 0 to i = 11, as initially the value of i is Jan which is 0 and the value of Dec is 11.

Interesting facts about initialization of enum.

1. **Two enum names can have same value**. For example, in the following C program both ‘Failed’ and ‘Freezed’ have same value 0.

```
```

```
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};

int main()
{
printf("%d, %d, %d", Working, Failed, Freezed);
return 0;
}
```

Output:

```
1, 0, 0
```

2. **If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0.** For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

Example:

```
#include <stdio.h>
enum day {sunday, monday, tuesday, wednesday, thursday, friday,
saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Output:

```
The day number stored in d is 4
```

3. **We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.**

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5,
               wednesday, thursday = 10, friday, saturday};

int main()
{
        printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,
                        wednesday, thursday, friday, saturday);
```

```
        return 0;
}
```

Output:

```
1 2 5 6 10 11 12
```

4. **The value assigned to enum names must be some integral constant**, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

5. **All enum constants must be unique in their scope**. For example, the following program fails in compilation.

```
enum state  {working, failed};
enum result {failed, passed};

int main()  { return 0; }
```

Output:

```
Compile Error: 'failed' has a previous declaration as 'state failed'
```

Difference Between malloc() and calloc()

The functions **malloc()** and **calloc()** are library functions that allocate memory dynamically. Dynamic means the memory is allocated during runtime (execution of the program) from the heap segment.

Initialization

malloc() allocates a memory block of given size (in bytes) and returns a pointer to the beginning of the block. malloc() doesn't initialize the allocated memory. If you try to read from the allocated memory without first initializing it, then you will invoke undefined behavior which will usually mean the values you read will be garbage.

calloc() allocates the memory and also initializes every byte in the allocated memory to 0. If you try to read the value of the allocated memory without initializing it, you'll get 0 as it has already been initialized to 0 by calloc().

Parameters

malloc() takes a single argument, which is the number of bytes to allocate

Unlike malloc(), calloc() takes two arguments:

1. **Number of blocks to be allocated.**
2. **Size of each block in bytes.**

Return Value

After successful allocation in malloc() and calloc(), a pointer to the block of memory is returned otherwise NULL is returned which indicates failure.

Example

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

int main()
{
        // Both of these allocate the same number of bytes,
        // which is the amount of bytes that is required to
        // store 5 int values.

        // The memory allocated by calloc will be
        // zero-initialized, but the memory allocated with
        // malloc will be uninitialized so reading it would be
        // undefined behavior.
        int* allocated_with_malloc = malloc(5 * sizeof(int));
        int* allocated_with_calloc = calloc(5, sizeof(int));

        // As you can see, all of the values are initialized to
        // zero.
        printf("Values of allocated_with_calloc: ");
        for (size_t i = 0; i < 5; ++i) {
                printf("%d ", allocated_with_calloc[i]);
        }
        putchar('\n');

        // This malloc requests 1 terabyte of dynamic memory,
        // which is unavailable in this case, and so the
        // allocation fails and returns NULL.
        int* failed_malloc = malloc(1000000000000);
        if (failed_malloc == NULL) {
                printf("The allocation failed, the value of "
                        "failed_malloc is: %p",
                        (void*)failed_malloc);
        }

        // Remember to always free dynamically allocated memory.
        free(allocated_with_malloc);
        free(allocated_with_calloc);
}
```

Output

```
Values of allocated_with_calloc: 0 0 0 0 0
The allocation failed, the value of failed_malloc is: (nil)
```

malloc()	calloc()
It is a function that creates one block of memory of a fixed size.	It is a function that assigns more than one block of memory to a single variable.

It only takes one argument.	It takes two arguments.
It is faster than calloc.	It is slower than malloc()
It has high time efficiency	It has low time efficiency
It is used to indicate memory allocation	It is used to indicate contiguous memory allcoation