Rajesh Parajuli

+9779847546279

Programming concept with C

Syllabus

Assignments

Assignment 1
Assignment 2
Assignment 3
Assignment 4

Study Materials

Unit 1
Unit 2
Unit 3
Unit 4
Unit 5
Unit 6
Unit 7
Unit 8

Online Compiler
Introduction of Programming Language
Assembler, Compiler and Interpreter
Syntax & Semantics
Programming Designing Tools
Features of Good Programme
Unit2 History of C Program
Basic Structure of C program
Character Set, Token & Comments
Variable
Datatypes
Type Conversion / Type Casting
Operators

Control structure
Selective Structure
Looping Structure
Nested Loop
Loop interrupts
Unit 4 Function
Function prototype, definition and call
Different ways of using function
Call by value call by reference
Recursion Function

Concept of array
Array declare, access and initialization
Multi-dimensional Array
Concept of Pointer
Pointer Address, deference, declaration, assignment, initialization
Pointer Arithmetic
Array and Pointer
Difference between Malloc and Calloc
String
String Function in C
Pointer and String

Unit 6 Structure and Union
Initializing, accessing member of structure
Array of structure
Pointer of structure
Union
Difference between structure and union
Unit 7 Concept of File handling
File Access Methods
Functions of file handling: fopen(), fclose(),fflush(), freopen()
Formatted input output
direct input output
Random File Access
Error handling
File operation

What is Program?

The program is the set of instructions that command the computer to perform a particular operation or a specific task.

- Instruction is a statement.
- A statement is an instruction to do only one task.
- A group of statements is composed together to form a program.

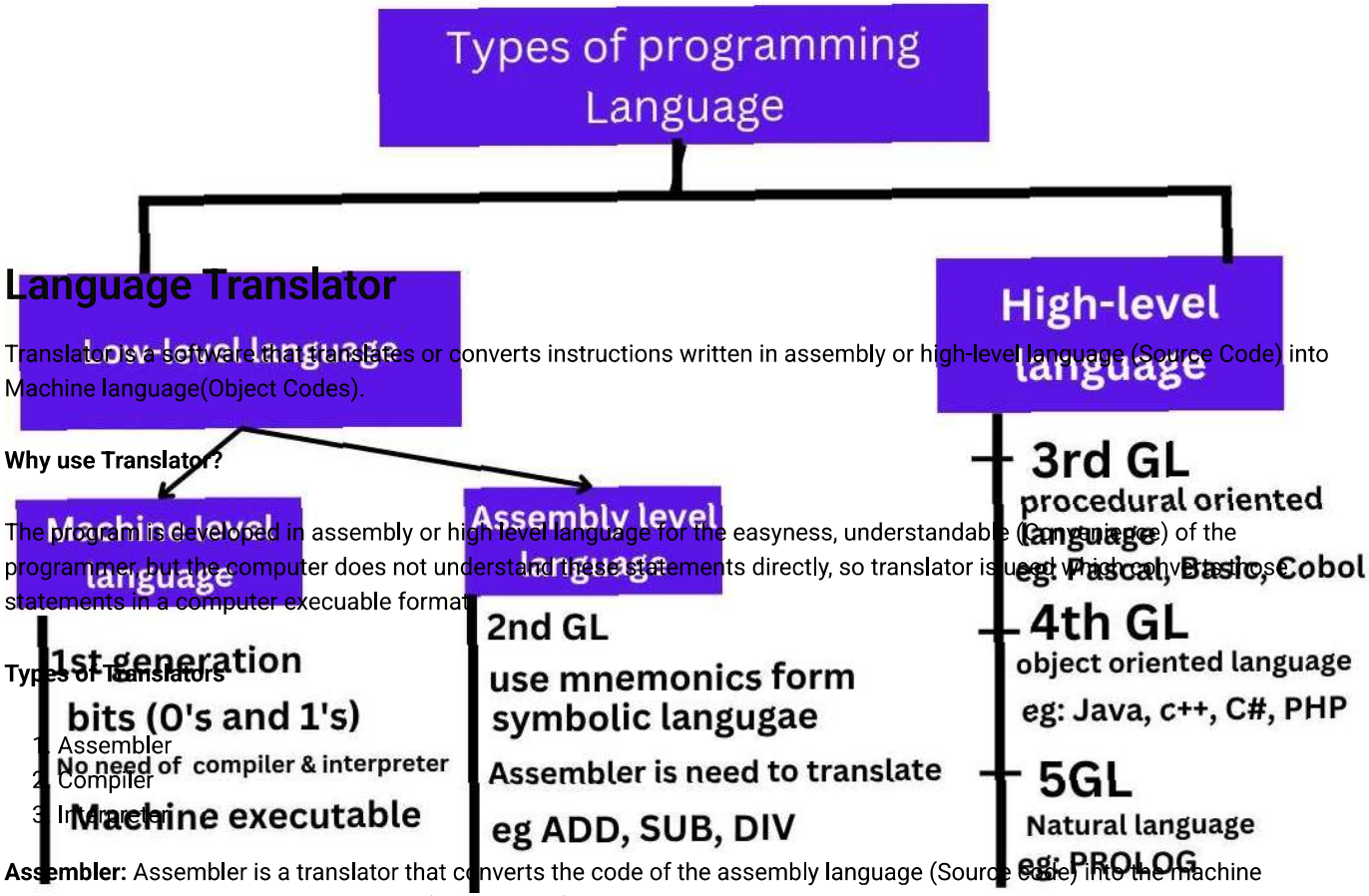
Unit 1 Introduction of Programming Language.

A programming language is a computer language that is used by programmers (developers) to communicate with computers. It is a set of instructions written in any specific language (C, C++, Java, Python) to perform a specific task.

- A programming language is a language that allows people to write specific commands to be executed on a computer.
- A programming language is mainly used to develop desktop applications, websites, and mobile applications.
- Most commonly used Programming Language are Python, JAVA, C, C++, C#, RUBY, PHP … etc.

Types of Programming Language

1. Low-level Language
2. High Level Language



Translator is a software that translates or converts instructions written in assembly or high-level language (Source Code) into Machine language(Object Codes).

Why use Translator?

The program is developed in assembly or high-level language for the easyness, understandable (Convenience) of the programmer but the computer does not understand these statements directly, so translator is used which converts these statements in a computer executable format.

Types of Translators

1. Assembler
2. Compiler
3. Interpreter

Assembler: Assembler is a translator that converts the code of the assembly language (Source code) into the machine executible language machine language (Object Code). When assembler completed it converted process then only it started to execute the program.

Compiler: A Compiler is a translator which translates the complete high level program (Source Code) into the machine code (object code) at once if the program doesn't contain syntax errors. Programming languages like C, C++, Java use the compiler.

Interpreter: An interpreter is a language translator which translates high-level language into the machine language at line at a time and executes the line of the program after it has been translated. It translates statements line by line.

Differences between Compiler and interpreter?

Compiler	Interpreter
Compiler translates the whole program into object code at a time.	Interpreter Translates one line or single statement of a program into object code at a time.

Translating process is faster	Translating process is slower
It stores the converted machine code from your source code program on the disk.	It never stores the machine code at all on the disk.
The compiler shows the complete errors and warning messages at program compilation time. So it is not possible to run the program without fixing program errors. Doing debugging of the program is comparatively complex while working with a compiler	An interpreter reads the program line-by-line; it shows the error if present at that specific line. You must have to correct the error first to interpret the next line of the program. Debugging is comparatively easy while working with an Interpreter.
Examples of compiler based programming language are C, C++, Java, COBOL, Pascal, FORTRAN	Examples of Interpreter based programming are BASIC, C#, PHP

Syntax

In a programming language, Syntax defines the rules that govern the structure and arrangement of keywords, symbols, and other elements. Syntax doesn't have any relationship with the meaning of the statement; it is only associated with the grammar and structure of the programming language.

- A line of code is syntactically valid and correct if it follows all the rules of syntax.
- Syntax does not have to do anything with the meaning of the statement.
- Syntax errors are easy to catch.
- Syntax errors are encountered after the program has been executed

Semantics

Semantics refers to the meaning of the associated line of code and how they are executed in a programming language. semantics helps interpret what function the line of code/program is performing.

- If there is any semantic error and even when the statement has correct syntax, it wouldn't perform the function that was intended for it to do. Thus, such errors are difficult to catch.
- Semantics are encountered at runtime.

Programming Design Tools

Program design Tools are the tools that are used to design a program before it actually developed. Program design tools are used by the developers. Some program design tools are: Algorith, Flow Charts, Pseudo Code, Data flow Diagram(DFD), Usecase Diagram... etc.

Algorithm: An algorithm is the sequence of steps that needs to be followed in order to acheive certain task. An algorithm is the fininte set of step by step set of statements that is used to solve a particular problem.

- It is written in simple human readable English Language.

An Algorithm should have the following properties:

1. It should have an input.
2. The steps mentioned in an algorithm can be executable by the computer
3. Each and every instruction should be in a simple language.
4. The number of steps should be finite.
5. It should not depend on a particular computer language or computer.
6. The algorithm should give an output after executing the finite numbers of steps.

Example of an algorithm:

1. Find the sum of two numbers.

step 1: START

Step2: Read the two numbers A and B.

Step3: Add the number A and B and store in S. or S=A+B.

Step4: Display D or Print D.

Step5: Stop.

2. Find the simple interest (SI).

Step1: Start.

Step2: Read the principal, rate, and time.

Step3: Multiply principal, rate and time.

Step4: Divide the product by 100 and store it in SI.

Step5: Print SI

Step6: END

3. Write an algorithm to Print 1 to 20 and also make flowchart.

Step1: Start.

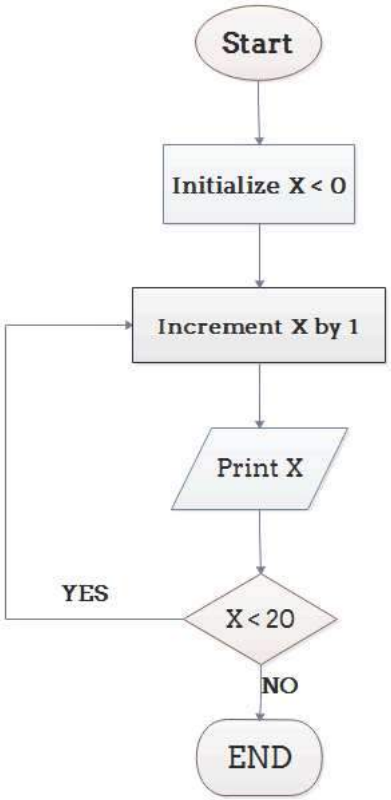
Step2: Initialize X as 0,

Step3: Increment X by 1,

Step4: Print X,

Step5: If X is less then 20 then repeat from step 2 until X=20.

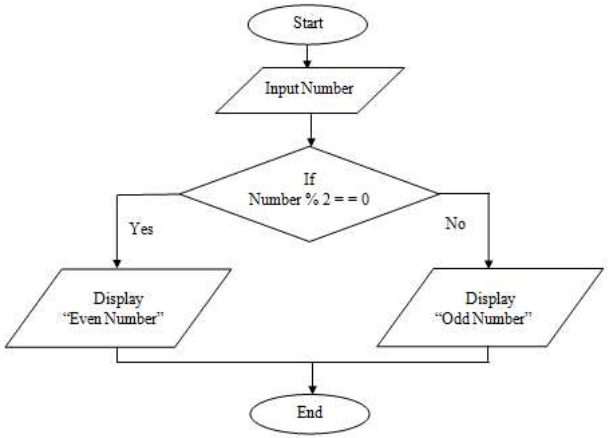
Step6: Stop.



Flowchart

A flowchart is a pictorial representation of an algorithm. **Flowchart** is a diagrammatic representation of sequence of logical steps of a program. Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowchart to find a given number is odd or even.



Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

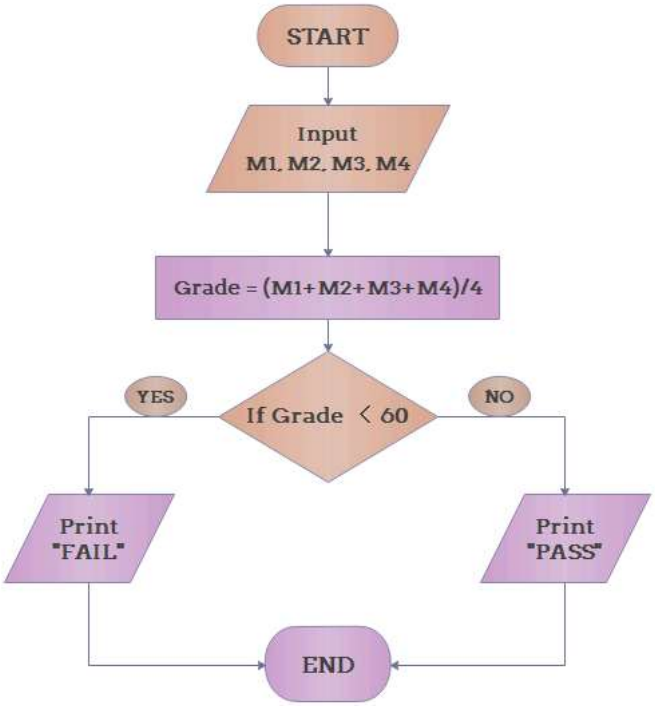
Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.

	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

4. Determine Whether A Student Passed the Exam or Not:

Algorithm

- Step 1: Input grades of 4 courses M1, M2, M3 and M4,
- Step 2: Calculate the average grade with formula “Grade=(M1+M2+M3+M4)/4”
- Step 3: If the average grade is less than 60, print “FAIL”, else print “PASS”.

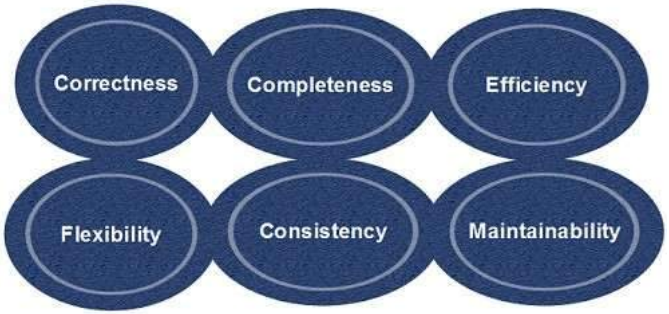


Differences between Algorithm and Flowchart?

Algorithm	Flowchart
It is a procedure for solving problems.	It is a graphic representation of a process.

The process is shown in step-by-step instruction.	The process is shown in block-by-block information diagram.
It is complex and difficult to understand.	It is intuitive and easy to understand.
It is convenient to debug errors.	It is hard to debug errors.
The solution is showcased in natural language.	The solution is showcased in pictorial format
It is somewhat easier to solve complex problem.	It is hard to solve complex problem.
It costs more time to create an algorithm.	It costs less time to create a flowchart.

Features of Good Programme.



1. **Correctness**: Program design should be correct as per requirement.
2. **Completeness**: The design should have all components like data structures, modules, and external interfaces, etc.
3. **Efficiency**: Resources should be used efficiently by the program.
4. **Flexibility**: Able to modify on changing needs.
5. **Consistency**: There should not be any inconsistency in the design.
6. **Maintainability**: The design should be so simple so that it can be easily maintainable by other designers.

Portability

A program should be supported by many different computers. The program should compile and run smoothly on different platforms. Because of rapid development in hardware and software, platform change is a common phenomenon these days. So, portability is measured by how a software application can be transferred from one computer environment to another without failure. A program is said to be more portable if it is easily adopted on different computer systems. Subsequently, if a program is developed only for a particular platform, its life expectancy is seriously compromised.

Maintainability

It is the process of fixing program errors and improving the program. If a program is easy to read and understand, then its maintenance will be easier. It should also prevent unwanted work so that the maintenance cost in the future will be low. It should also have quality to easily meet new requirements. A maintainable software allows us to fix bugs quickly and easily, improve usability and performance, add new features, make changes to support multiple platforms, and so on.

Efficient

Program is said to be more efficient if it takes the least amount of memory and processing time and is easily converted to machine language. The algorithm should be more effective. Every program needs a certain amount of processing time and memory to process the instructions and data. The program efficiency is also high if it has a high speed during runtime execution of the program.

Reliable

The user's actual needs will change from time-to-time, so the program is said to be reliable if it works smoothly in every version. It is measured as reliable if it gives the same performance in all simple to complex conditions.

Machine Independent

Program should be machine-independent. Program written on one system should be able to execute on many different types of computers without any changes. It is not system specific and provides more flexibility. An example of this would be Java.

Cost Effectiveness

Cost Effectiveness is the key to measure the program quality. The cost must be measured over the life of the program and must include both costs and human costs of producing these programs.

Flexible

The program should be written in such a manner that it allows one to add new features without changing the existing module. The majority of the projects are developed for a specific period, and they require modifications from time to time. It should always be ready to meet new requirements. Highly flexible software is always ready for a new world of possibilities.

Unit-2 Introduction to C

- C programming is a general-purpose, procedural programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system.
- The UNIX OS was totally written in C.

History of C

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A. **Dennis Ritchie** is known as the **founder of the c language**. C was developed to overcome the problems of

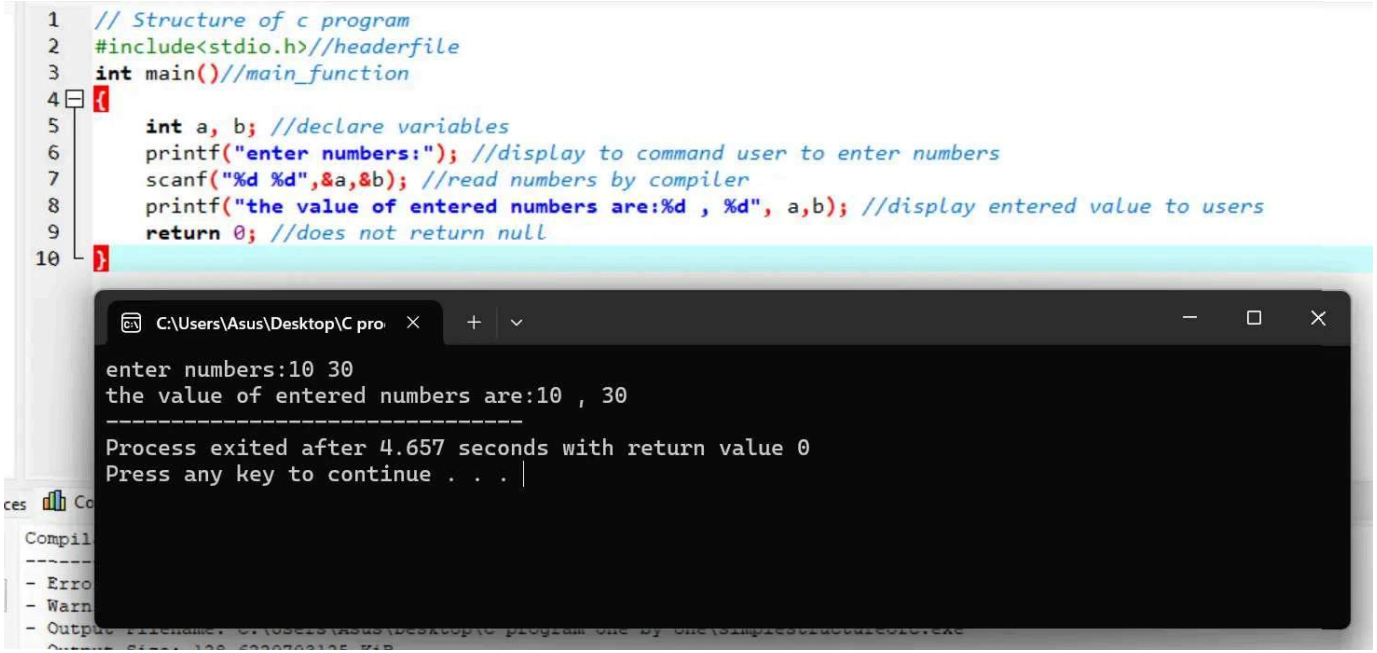
previous languages such as B, BCPL, etc.

Initially, C language was developed to be used in UNIX operating system. It inherits many features of previous languages such as B and BCPL.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie

Today C is the most widely used and popular System Programming Language. Today's most popular Linux OS and RDBMS MySQL have been written in C.

Structure of C Program



1. Documentation (Documentation Section)
2. Preprocessor Statements (Link Section)

3. Definition Section
4. Global Declarations Section
5. Main functions section
6. User-Defined Functions or Sub program functions

In C language, all these six sections together make up the Basic Structure of C Program

1. Documentation (Documentation Section)

Programmers write comments in the Documentation section to describe the program. The compiler ignores the comments and does not print them on the screen. Comments are used only to describe that program.

```
/* File Name -: Hello.c
   Author Name -: Rajesh parajuli Founder of parajulirajesh.com.np
   Date -: 12/09/2023
   Description -: Basic Structure of C program */
//This is a single line comment
```

2. Preprocessor Statements (Link Section)

Within the Link Section, we declare all the Header Files that are used in our program. From the link section, we instruct the compiler to link those header files from the system libraries, which we have declared in the link section in our program.

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

In addition to all these Header Files in the Link Section, there are a lot of Header Files which we can link in our program if needed.

3. Definition Section

The definition of Symbolic Constant is defined in this section, so this section is called Definition Section. Macros are used in this section.

```
#define PI 3.14
```

4. Global Declarations Section

Within the Global Declarations Section, we declare such variables which we can use anywhere in our program, and that variable is called Global Variables, we can use these variables in any function.

In the Global Declaration section, we also declare functions that we want to use anywhere in our program, and such functions are called Global Function.

```
int area (int x);  //global function
int n;            // global Variable
```

5. Main functions section

Whenever we create a program in C language, there is one main() function in that program. The main () function starts with curly brackets and also ends with curly brackets. In the main () function, we write our statements.

The code we write inside the main() function consists of two parts, one Declaration Part and the other Execution Part. In the Declaration Part, we declare the variables that we have to use in the Execution Part, let's understand this with an example.

```
int main (void)
{
 int n = 15; // Declaration Part
 printf ("n = %d", n); // Execution Part
 return (0);
}
```

6. User-Defined Functions or Sub Program Section

Declare all User-Defined Functions under this section.

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

What is the structure of C program syntax?

Any C Program can be divided into header, main() function, variable declaration, body, and return type of the program.

Basic Structure explaining with an example.

```
// Documentation
/**
* file: sum.c
* author: you
* description: program to find sum.
*/

// Link
#include <stdio.h>

// Definition
#define X 20

// Global Declaration
int sum(int y);

// Main() Function
int main(void)
{
int y = 55;
printf("Sum: %d", sum(y));
return 0;
}

// Subprogram
int sum(int y)
{
return y + X;
}
```

Sections	Description
/ * file: sum.c * author: you * description: program to find sum. */**	It is the comment section and is part of the description section of the code.
#include<stdio.h>	Header file which is used for standard input-output. This is the preprocessor section.
#define X 20	This is the definition section. It allows the use of constant X in the code.
int sum(int y)	This is the Global declaration section includes the function declaration that can be used anywhere in the program.
int main()	main() is the first function that is executed in the C program.
{...}	These curly braces mark the beginning and end of the main function.
printf("Sum: %d", sum(y));	printf() function is used to print the sum on the screen.
return 0;	We have used int as the return type so we have to return 0 which states that the given program is free from the error and it can be exited successfully.
int sum(int y) {	This is the subprogram section. It includes the user-defined functions that are called in the main() function.

Sections	Description
return y + X; Write a C program to implement subtraction by giving two numbers from user inputs. **}**	

```
//C Program To Subtract Two Numbers
#include<stdio.h>
int main()

{
int num1, num2, difference;

//Asking for input
printf("Enter first number: ");
scanf("%d", &num1);
printf("Enter second number: ");
scanf("%d", &num2);

difference = num1 – num2;
printf("Difference of num1 and num2 is: %d",difference);
return 0;
}
```

Character Set

As every language contains a set of characters used to construct words, statements, etc., C language also has a set of characters which include **alphabets, digits**, and **special symbols**. C language supports a total of 256 characters. Every character in C language has its equivalent ASCII (American Standard Code for Information Interchange) value.

Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set. C language character set contains the following set of characters

1. Alphabets
2. Digits
3. Special Symbols

Alphabets:

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

lower case letters – **a to z**

UPPER CASE LETTERS – **A to Z**

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

Digits – **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
Special Symbols:

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

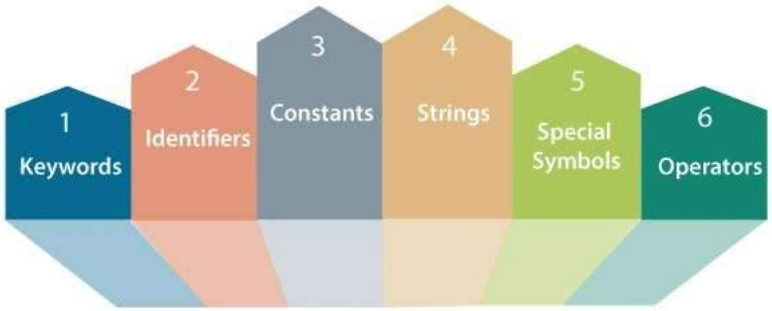
Special Symbols – **~ @ # $ % ^ & * ( ) _ – + = { } [ ] ; : ‘ ” / ? . > , < \ |**

Symbols	Name	Symbols	Name
~	Tilde	>	Greater than
<	Less than	&	Ampersand
|	Or/pipe	#	Hash
>=	Greater than equal	<=	Less than equal
==	Equal	=	Assignment
!=	Not equal	^	Caret
{	Left brace	}	Right brace
(	Left parenthesis	)	Right parenthesis
[	Left square bracket	]	Right square bracket
/	Forward slash	\	Backward slash
:	Colon	;	Semicolon
+	Plus	-	Minus
*	Multiply	/	Division
%	Mod	,	Comma
‘	Single quote	“	Double quote
>>	Right shift	<<	Left shift
.	Period	_	Underscore

Token in C

A token is a smallest individual element of a program which is meaningful to the compiler. The compiler that breaks a program into the smallest units is called tokens and these tokens proceed to the different stages of the compilation.

- Tokens in C are building blocks which means a program can’t be created without tokens.



Classification of C Tokens

1.Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier.

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

C predefined Keywords

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

2.Identifiers

Identifiers in C are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Identifiers cannot be used as keywords. Rules for constructing identifiers in C are given below:

- An identifier can only have alphanumeric characters (a-z , A-Z , 0-9) (i.e. letters and digits) and underscore(_) symbol.
- Identifier names must be unique.
- The first character must be an alphabet or underscore.
- You cannot use a keyword as an identifier.

- Only the first thirty-one (31) characters are significant.
- It must not contain white spaces.
- Identifiers are case-sensitive.

For Example:

int cprogram;

Char Bicte_firstsemester;

here, int and Char is keywords and cprogram and Bicte_firstsemester is identifier.

3.Constants

The constants in C are the read-only variables whose values cannot be modified once they are declared in the C program. The type of constant can be an integer constant, a floating pointer constant, a string constant, or a character constant. In C language, the **const** keyword is used to define the constants.

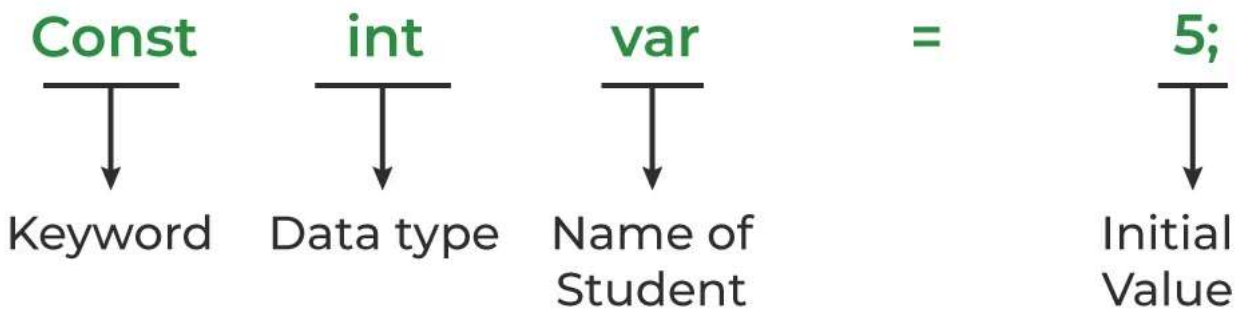
What is a constant in C?

As the name suggests, a constant in C is a variable that cannot be modified once it is declared in the program. We can not make any change in the value of the constant variables after they are defined.

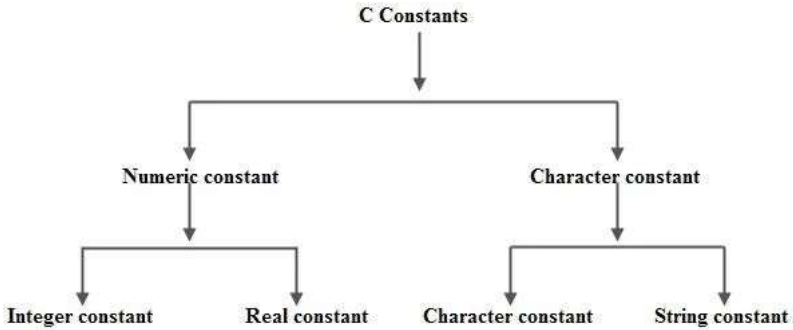
Syntax to Define Constant

```
const data_type var_name = value;
```

Constants



Types of Constants in C




```
// C program to illustrate constant variable definition
#include <stdio.h>

int main()
{

// defining integer constant using const keyword
const int int_const = 25;
```

```
// C Program to demonstrate the behaviour of constant
// variable
#include <stdio.h>

int main()
{
// declaring a constant variable
const int var;
// initializing constant variable var after declaration
```

```
// defining character constant using const keyword
const char char_const = 'A';

// defining float constant using const keyword
const float float_const = 15.66;

printf("Printing value of Integer Constant: %d\n",
int_const);
printf("Printing value of Character Constant: %c\n",
char_const);
printf("Printing value of Float Constant: %f",
float_const);

return 0;
}
```

```
var = 20;

printf("Value of var: %d", var);
return 0;
}
```

4.Strings

- Sequence of Characters is known as Strings.
- Every String is terminated by \0
- String Constant is a sequenced os 0 or more characters enclosed between double quotes " " is known as string constant e.g. "S", "XYZ", "123", "hello world"
- All characters are converted into their corresponding ASCII value and then stored in memory as contiguous allocation.
- String Variable is the array of character type. For e.g. char[10];

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

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

```
char greeting[] = "Hello";
```

```
#include <stdio.h>

int main () {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\
```

```
#include <stdio.h>
int main()
{
char name[6];
printf("Enter name: ");
scanf("%s", name);
```

```
    printf("Greeting message: %s\n", greeting );
    return 0;
}
```

```
printf(“Your name is %s.”, name);
return 0;
}
```

Like many other programming languages, strings in C are enclosed within ***double quotes(" ")***, whereas characters are enclosed within ***single quotes(' ')***. When the compiler finds a sequence of characters enclosed within the double quotation marks, it adds a null character (**\0**) at the end by default.

1. Character arrays are used for declaring strings in C.
2. The general syntax for declaring them is:

char variable[array_size];

5.Special Symbols

Special Symbols are symbols in C language that have special meaning and can not be used for any other purpose.

- **Brackets[]:** Opening and closing brackets are used as array element references. These indicate single and multidimensional subscripts.
- **Parentheses():** These special symbols are used to indicate function calls and function parameters.
- **Braces{}:** These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.
- **Comma (,):** It is used to separate more than one statement like for separating parameters in function calls.
- **Colon(:):** It is an operator that essentially invokes something called an initialization list.
- **Semicolon(;):** It is known as a statement terminator. It indicates the end of one logical entity. That's why each individual statement must be ended with a semicolon.
- **Asterisk (*):** It is used to create a pointer variable and for the multiplication of variables.
- **Assignment operator(=):** It is used to assign values and for logical operation validation.
- **Pre-processor (#):** The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.
- **Period (.):** Used to access members of a structure or union.
- **Tilde(~):** Used as a destructor to free some space from memory.

Square Brackets [ ]
The opening and closing square brackets represent single and multi-dimensional subscripts and they are used as array element reference for accessing array elements.
int arr[10]; //For declaring array, with size defined in square brackets

Simple Brackets ( )
The opening and closing circular brackets are used for function calling and function declaration.

get_area(100); //Function calling with 100 as parameter passed in circular brackets

Curly Braces { }

In C language, the curly braces are used to mark the start and end of a block of code containing executable logical statements.

```
int main{
printf("Illustrating the use of curly braces!");
}
```

Comma (,)

Commas are used to separate variables or more than one statement just like separating function parameters in a function call.

```
int a=10,b=20,c=30; //Use of comma operator
```

Pre-Processor / Hash (#)

It is a macro-processor that is automatically used by the compiler and denotes that we are using a header file.

```
#include<stdio.h> //For defining header-file
#define ll long

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

Asterisk (*)

Asterisk symbols can be used for multiplication of variables and also for creating pointer variables. **Example:**

```
int main(){
int a = 20,b = 10;
int sum = a*b; //Use of asterisk in multiplication
int *ptr = &a;
//Pointer variable ptr pointing to address of integer variable a
}
```

Tilde (~)

It is used as a destructor to free some space from the memory.

```
int main(){
int n = 2;
printf("Bitwise complement of %d: %d", n, ~n);
//Bitwise complement of 2 can be found with the help of tilde operator and the result here is -3
}
```

Period (.)

It is used to access members of a structure or a union.

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

```
struct Person { //structure defined
int city_no; //members of structure
float salary;
}person1;

int main(){
person1.city_no = 100;
//accessing members of structure using period (.) operator
person1.salary = 200000;
printf("City_Number: %d",person1.city_no);
printf("\nSalary: %.2f",person1.salary);
return 0;
}
```

Colon (:)
It is used as a part of conditional operator **(? :)** in C language.
Example:

```
int a = 10,b = 20,c;
c = (a < b) ? a : b;
//If a<b is true, then c will be assigned with the value of a else b
printf("%d", c);
```

Semicolon (;)
It is known as a statement terminator and thus, each logical statement of C language must be ended with a semi-colon.
Example:

```
int a=10; //Semi-colon is widely used in C programs to terminate a line
```

Assignment Operator (=)
It is used to assign values to a variable and is sometimes used for logical operation validation.
Example:

```
int a = 10, b = 20; //Assignment operator is used to assign some values to the variables
```

6.operators

Operators are symbols that are used to perform some operation or a set of operations on a variable or a set of variables. C has a set of operators to perform specific mathematical and logical computations on operands. C Supports a rich set of built-in Operators. Operators are used to perform operations on variables and values.

1. On the Basis of number of operands required for an operator

Example: Y= a+b

Unary Operator: The operator which require only one operand are known as unary Operator. Examples: ++ (increment operator), — (decrement operator). X++, Y–

- –, + are operator
- a, b are operand

Binary Operators: The operators which require two operands are known as binary operators. For example: A+B, A-B, A*B, A/B etc.

Types of Operator:

Ternary Operators: The operators that require three operands are known as ternary operators. For Example: A ? B : C (this is also a condiotional operator).

1. on the basis of the number of operands required for an operator
2. On the basis of utility or functions of an operator

2. On the Basis of Utility (or functions) of an operator

According to the utility and action, operators are classified into following categories:

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

Arithmetic Operators

Arithmetic Operators are the type of operators in C that are used to perform mathematical operations in a C program. They can be used in programs to define expressions and mathematical formulas.

Operator	Name of the Operator	Arithmetic Operation	Syntax
+	Addition	Add two operands.	x + y
–	Subtraction	Subtract the second operand from the first operand.	x – y
*	Multiplication	Multiply two operands.	x * y
/	Division	Divide the first operand by the second operand.	x / y

Operator	Name of the Operator	Arithmetic Operation	Syntax
%	Modulus division	Calculate the remainder when the first operand is divided by the second operand.	x % y

```
//C program to demonstrate arithmetic operators
#include <stdio.h>

int main()
{
int a = 10, b = 4, res;

// printing a and b
printf(“a is %d and b is %d\n”, a, b);

res = a + b; // addition
printf(“a + b is %d\n”, res);

res = a – b; // subtraction
printf(“a – b is %d\n”, res);

res = a * b; // multiplication
printf(“a * b is %d\n”, res);

res = a / b; // division
printf(“a / b is %d\n”, res);

res = a % b; // modulus
printf(“a %% b is %d\n”, res);

return 0;
}
```

Relaional/ Comparison Operators

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

Operator	Name	Example	Result
==	Equal to	x == y	Returns 1 if the values are equal
!=	Not equal	x != y	Returns 1 if the values are not equal

>	Greater than	x > y	Returns 1 if the first value is greater than the second value
<	Less than	x < y	Returns 1 if the first value is less than the second value
>=	Greater than or equal to	x >= y	Returns 1 if the first value is greater than, or equal to, the second value
<=	Less than or equal to	x <= y	Returns 1 if the first value is less than, or equal to, the second value

```
// C program to demonstrate working of relational operators
#include <stdio.h>

int main()
{
int a = 10, b = 4;

// greater than example
if (a > b)
printf(“a is greater than b\n”);
else
printf(“a is less than or equal to b\n”);

// greater than equal to
if (a >= b)
printf(“a is greater than or equal to b\n”);
else
printf(“a is lesser than b\n”);

// less than example
if (a < b)
printf(“a is less than b\n”);
else
printf(“a is greater than or equal to b\n”);

// lesser than equal to
if (a <= b)
printf(“a is lesser than or equal to b\n”);
```

```
else
printf("a is greater than b\n");

// equal to
if (a == b)
printf("a is equal to b\n");
else
printf("a and b are not equal\n");

// not equal to
if (a != b)
printf("a is not equal to b\n");
else
printf("a is equal b\n");

return 0;
}
```

Logical operators

Logical Operators are used to compare or evaluate logical and relational expressions. The operands of logical operators must be either Boolean value (1 or 0) or expressions that produces Boolean value. The Output of these operators is always either 1 true or 0 False. The logical Operators supported in C are :

- && logical AND : it produces true if each operand is true otherwise it produces false.
- || Logical OR : it produces true when any of the conditions is true.
- ! Logical NOT- it reverse to the operand.

Write a program to illustrate the output of logical operators.

```
#include<stdio.h>
#include<conio.h>
int main()
{
int a=10, b=5, c=40;
printf("a<b && a<c is %d\n", (a<b && a<c));
printf("a>b && b>c is %d\n", (a>b && b<c));
printf("a<b || a<c is %d\n", (a<b || a<c));
printf("a>b || b<c is %d\n", (a>b || b<c));
printf("a>c || b>c is %d\n", (a>c || b>c));

Printf("not operator ", a!b);
```

```
getch();
return 0;
}
```

Assignment operators

Assignment Operators are also binary operators and they are used to assign result of an expression to a variable. The mostly used assignment operator is '='. There are other shorthand assignment operators supported by C. They are +=, -=, *=, /+ and %=. These Operators are also known as arithmetic assignment operators.

+= Addition Assignment (a+=b, means a=a+b) assign sum of a and b to a.

-= Subtraction Assignment (a-=b, means a=a-b) assign subtraction of a and b to a)

= Multiplication Assignment (a=b, means a=a*b) assign multiplication of a and b to a)

/= Division Assignment (a/=b, means a=a/b) assign division of a and b to a)

%= Remainder Assignment (a%=b, means a=a%b) assign remainder of a divisible by b to a)

Demostrate the assignment operator in c program

```
#include<stdio.h>
#include<conio.h>
int main()
{
int a=10, b=5;
b+=a; //b=b+a
printf("b=%d", b);
getch();
return 0;
}
```

Output:

```
b=15
```

Increment and decrement operators

The increment operator is used to increase the value of an operand by 1; and the decrement operator is used to decrease the value of an operand by 1. They take one operand, so called unary operator. The syntax for the operator is:

- ++ variable
- variable++
- –variable
- variable++

```
#include <stdio.h>
int main() {
int x = 10;
printf("Initial value of x: %d\n", x);
x++; // Increment
printf("After increment (x++): %d\n", x);
x–; // Decrement
printf("After decrement (x–): %d\n", x);
return 0;
}
```

Conditional Operators

The Operator named "?:" is known as conditional Operator. It takes three operands. Thus, it is also called ternary operator. The syntax is :

value= expression ? expression2: expression3

working principal

- If(expression 1)
- variable = expression2;
- else
- variable = expression3;

Write a program to read two numbers from user and determine the larger number using conditional operator.

```
#include<stdio.h>
#include<conio.h>
int main()
{
 int n1, n2, larger;
 printf("Enter two numbers:");
 scanf("%d%d", &n1, &n2);
 larger = n1>n2 ? n1: n2;
 Printf("The larger number is %d", larger);
```

```
 getch();
 return 0;
}
```

Output:

```
 Enter two numbers: 35 90
 The larger number is 90
```

Bitwise Operator

The bitwise operators are the operators used to perform the operations on the data at the bit-level. When we perform the bitwise operations, then it is also known as bit-level programming. It consists of two digits, either 0 or 1. It is mainly used in numerical computations to make the calculations faster. It can be used only integer type values not float, double etc.

We have different types of bitwise operators in the C programming language. The following is the list of the bitwise operators:

Operator	Meaning of operator
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR operator
~	One's complement operator (unary operator)
<<	Left shift operator
>>	Right shift operator

Let's look at the truth table of the bitwise operators.

X	Y	X&Y	X|Y	X^Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise exclusive operator gives 0 if both corresponding values are same. if both corresponding vales are not same , it gives 1.

Write a program to demostrate bitwise operator.

```
#include<stdio.h>
int main()
{
int a=7,b=14;
printf("Bitwise AND %d\n", a&b);
printf("Bitwise OR %d\n", a|b);
printf("Bitwise XOR %d\n", a^b);
return 0;
}
```

Output:

```
Bitwise AND 6
Bitwise OR 15
Bitwise XOR 9
```

Left Shift Operator

The left shift operator is a type of Bitwise shift operator, which performs operations on the binary bits. It is a binary operator that requires two operands to shift or move the position of the bits to the left side and add zeroes to the empty space created at the right side after shifting the bits.

Bitwise Left shift operator is used to shift the binary sequence to the left side by specified position.

Example

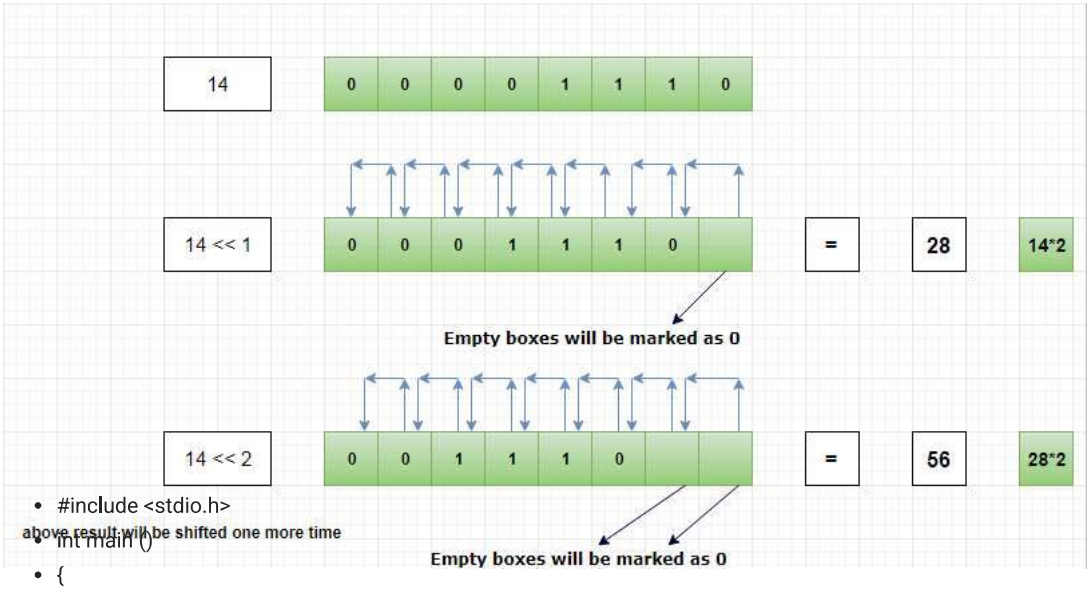
Let's take a number 14.

Binary representation of 14 is 00001110 (for the sake of clarity let's write it using 8 bit)

$14 = (00001110)_2$

Then 14 << 1 will shift the binary sequence 1 position to the left side.

Like,



- #include <stdio.h>
- int main ()
- {
- // declare local variable
- int num;
- printf (" Enter a positive number: ");
- scanf (" %d", &num);
- // use left shift operator to shift the bits
- num = (num << 2); // It shifts two bits at the left side
- printf (" \n After shifting the binary bits to the left side. ");
- printf (" \n The new value of the variable num = %d", num);
- return 0;
- }

```
Enter a positive number: 14
After shifting the binary bits to the left side.
The new value of the variable num = 56
```

In general, if we shift a number by n position to left, the output will be number * (2^n).

Right Shift Operator

The right shift operator is a type of bitwise shift operator used to move the bits at the right side, and it is represented as the double (>>) arrow symbol. Like the Left shift operator, the Right shift operator also requires two operands to shift the bits at the right side and then insert the zeroes at the empty space created at the left side after shifting the bits.

Bitwise Right shift operator >> is used to shift the binary sequence to right side by specified position.

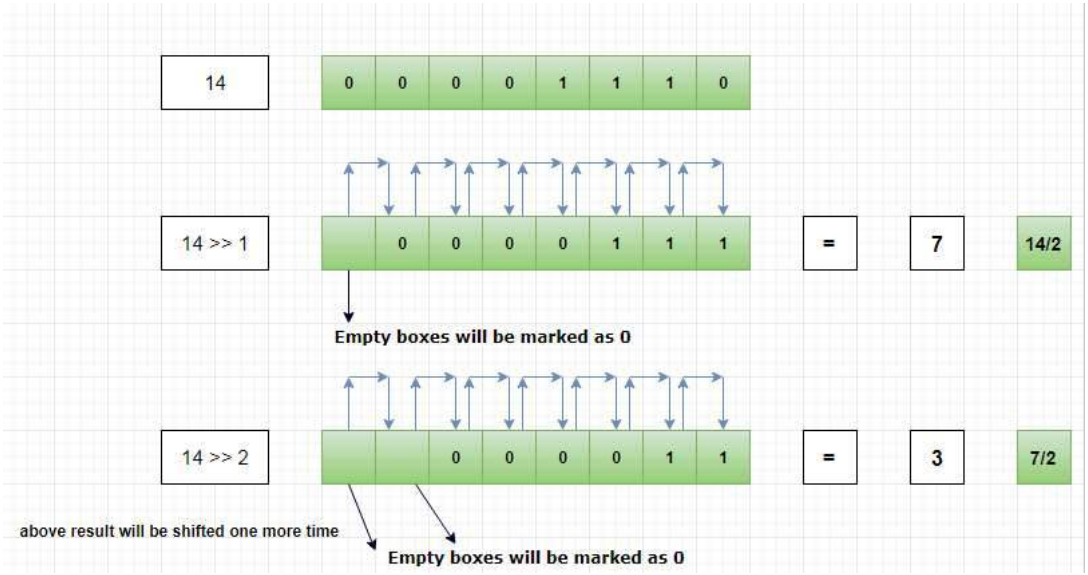
Example

Let's take a number 14.

Binary representation of 14 is 00001110 (for the sake of clarity let's write it using 8 bit)

$14 = (00001110)_2$

Then 14 >> 1 will shift the binary sequence by 1 position to the right side.



In general, if we shift a number by n times to right, the output will be number / (2^n) .

```
1. #include <stdio.h>
2. int main ()
3. {
4. // declare local variable
5. int num;
6. printf (" Enter a positive number: ");
7. scanf (" %d", &num);
```

```
8. // use right shift operator to shift the bits
9. num = (num >> 2); // It shifts two bits at the right side
10. printf (" \n After shifting the binary bits to the right side. ");
11. printf (" \n The new value of the variable num = %d", num);
12. return 0;
13. }
```

Output:

```
Enter a positive number: 14

After shifting the binary bits to the right side.
The new value of the variable num = 3
```

Datatypes

Datatypes refers to the types of data.

Category	Data Type	Format Specifier	Size (Bytes)	Range	Description
Primitive	char	%c	1	-128 to 127 (signed) / 0 to 255 (unsigned)	Stores a single character.
	signed char	%c	1	-128 to 127	Stores a signed character.
	unsigned char	%c	1	0 to 255	Stores an unsigned character.
	int	%d, %i	2 or 4	-32,768 to 32,767 (2-byte) / -2,147,483,648 to 2,147,483,647 (4-byte)	Stores integers (whole numbers).
	unsigned int	%u	2 or 4	0 to 65,535 (2-byte) / 0 to 4,294,967,295 (4-byte)	Stores non-negative integers.

	short int	%hd	2	-32,768 to 32,767	Stores small integer values.
	unsigned short int	%hu	2	0 to 65,535	Stores small unsigned integers.
	long int	%ld	4 or 8	-2,147,483,648 to 2,147,483,647 (4-byte) / -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (8-byte)	Stores long integers.
	unsigned long int	%lu	4 or 8	0 to 4,294,967,295 (4-byte) / 0 to 18,446,744,073,709,551,615 (8-byte)	Stores long unsigned integers.
	long long int	%lld	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Stores very large integers.
	unsigned long long int	%llu	8	0 to 18,446,744,073,709,551,615	Stores very large unsigned integers.
	float	%f	4	3.4E-38 to 3.4E+38	Stores single precision floating-point numbers.
	double	%lf	8	1.7E-308 to 1.7E+308	Stores double precision floating-point numbers.
	long double	%Lf	10, 12, or 16	3.4E-4932 to 1.1E+4932	Stores extended precision floating-point numbers.

	void	N/A	0	N/A	Represents no value or an unknown type.
Derived	Array	N/A	Depends	Depends on the type and number of elements	A collection of elements of the same type.
	Pointer	%p	4 or 8	Depends on the system architecture	Stores the memory address of another variable.
User-Defined	struct	N/A	Depends	Depends on the structure members	A collection of variables of different data types under one name.
	union	N/A	Depends	Depends on the largest member	Shares memory among its members, with only one member being used at a time.
	enum	%d	4	Based on the integer values assigned to constants	Represents a set of named integer constants.

Variable

Variable is a container that holds the value of any kind of data type. It is a case sensitive in c program. It is an identifier which store the value and reserved some memory space for data of any type.

Syntax:

data_type variable_name=value;

Rules for defining variable

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.

- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Example

```
#include <stdio.h>

int main() {
  int x = 5;// x is a variable of integer type
  char y[12] = “helloworld”;// y is a variable of string type
  char z=’b’; // z is a variable of character type
  float a=12.5; // a is a variable of float type
  double b=19.99; // b is a variable of double type
  printf(“%d\n%s\n%c\n%f\n%lf\n”, x,y,z,a,b);
  return 0;
}
```

Output:

```
5
helloworld
b
12.500000
19.990000
```

Datatypes List

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
`boolean`	1 bit	Stores true or false values
`char`	2 bytes	Stores a single character/letter or ASCII values

Type Casting / Type Conversion

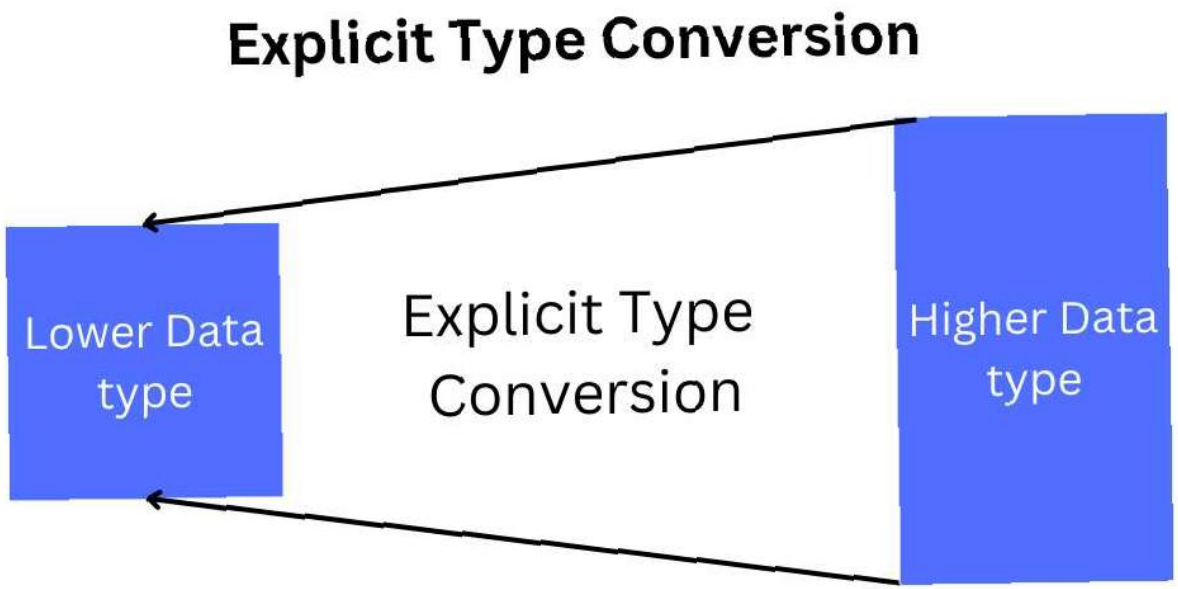
Type conversion is the way of converting a datatype of one variable to another datatype.

Types of type conversion:

1. Implicit type conversion- conversion is done by the compiler and there is no loss of information. It is automatic type conversion.
2. Explicit type conversion- conversion is done by the programmer and there will be loss of information.

Implicit Type Conversion

```
// An example of implicit conversion
#include <stdio.h>
int main()
{
int x = 10; // integer x
char y = 'a'; // character c

// y implicitly converted to int. ASCII
// value of 'a' is 97
x = x + y;

// x is implicitly converted to float
float z = x + 1.0;

printf("x = %d, z = %f", x, z);
return 0;
}
```



This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type

Syntax:

(type) expression

Example:

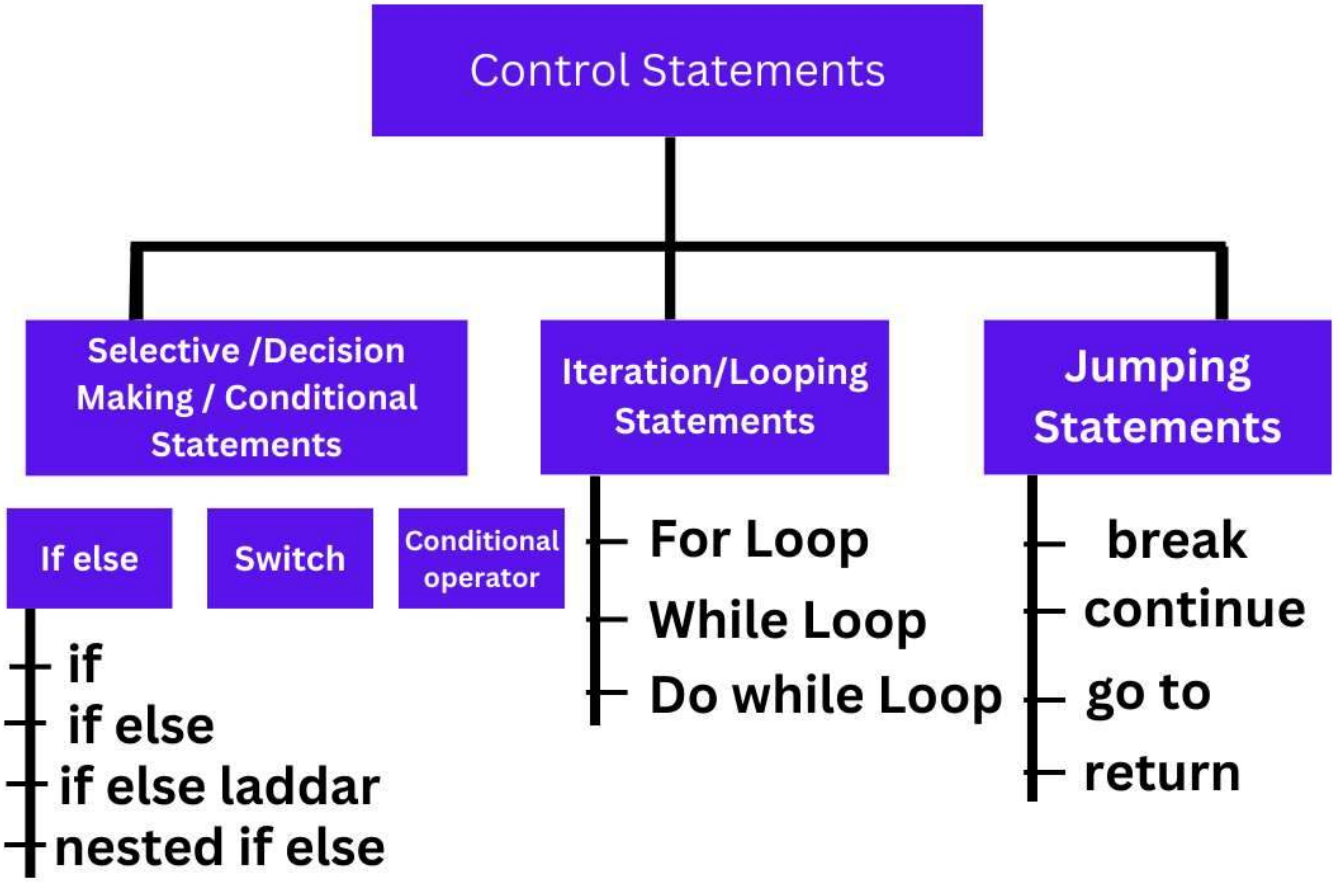
```
float f = 10.5;
int num = (int)f; // Explicit typecasting (float to int)
```

Control Statements in c

```
// C program to demonstrate explicit type casting
#include<stdio.h>
```

Control Statements control the flow of execution of the statements of a program. The various types of Control statements are :

```
int main()
{
```



1. Conditional Statements / Selective / Desision Structure

In conditional control , the execution of statements depends upon the condition-test. If the condition evaluates to true, then a set of statements is executed otherwise another set of statements is followed. This control is also called Decision Control or selective control statement because it helps in making decision about which set of statements is to be executed.

If statement: This is the most simple form of decision control statement. In this form, a set of statements are executed only if the condition given with if evaluates to true.

Syntax:

```
if(condition)
{
//if block of Statements executed if the given if condition is true ;
}
```

If else Statement: This is a bi-directional control statement. This statement is used to test a condition and take one of the two possible actions. If the condition evaluates to true then one statement (or block of statements) is executed otherwise other statement (or block of statements) is executed.

Syntax:

```
if(expression)
{
//code to be executed if condition is true
}
else
{
//code to be executed if condition is false
}
//executed outer statements;
```

If else ladder statements(if-else-if):

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks possib

Syntax:

```
if(condition1)
{
//code to be executed if condition1 is true
}
else if(condition2)
{
//code to be executed if condition2 is true
}
else if(condition3)
{
//code to be executed if condition3 is true
}
...
else
```

```
{
//code to be executed if all the conditions are false
}
//executed outer statements if it is there;
```

Nested if else: Nested if else statement is a control statement where both if else statement is there with having another if else statement inside it.

Syntax of nested if else:

```
if (condition1){
if (condition2)
stmt1;
else
stmt2;
}
else {
if (condition3)
stmt3;
else
stmt4;
}
```

Q. Find greatest number among three numbers using if statement.

```
#include <stdio.h>

int main() {

int n1, n2, n3;

printf("Enter three different numbers: ");
scanf("%d %d %d", &n1, &n2, &n3);

// if n1 is greater than both n2 and n3, n1 is the largest
if (n1 >= n2 && n1 >= n3)
printf("%d is the largest number.", n1);
```

Q. Find greatest number among three numbers using if else laddar statement.

```
#include <stdio.h>

int main() {

int n1, n2, n3;

printf("Enter three numbers: ");
scanf("%d %d %d", &n1, &n2, &n3);

// if n1 is greater than both n2 and n3, n1 is the largest
if (n1 >= n2 && n1 >= n3)
printf("%d is the largest number.", n1);
```

Q.Find the greatest number among three numbers using nested if else statement.

```
#include <stdio.h>

int main() {

int n1, n2, n3;

printf("Enter three numbers: ");
scanf("%d %d %d", &n1, &n2, &n3);

// outer if statement
if (n1 >= n2) {

// inner if...else
if (n1 >= n3)
```

```
// if n2 is greater than both n1 and n3, n2 is the largest
if (n2 >= n1 && n2 >= n3)
printf("%d is the largest number.", n2);

// if n3 is greater than both n1 and n2, n3 is the largest
if (n3 >= n1 && n3 >= n2)
printf("%d is the largest number.", n3);

return 0;
}
```

```
// if n2 is greater than both n1 and n3, n2 is the largest
else if (n2 >= n1 && n2 >= n3)
printf("%d is the largest number.", n2);

// if both above conditions are false, n3 is the largest
else
printf("%d is the largest number.", n3);

return 0;
}
```

```
printf("%d is the largest number.", n1);
else
printf("%d is the largest number.", n3);
}

// outer else statement
else {

// inner if...else
if (n2 >= n3)
printf("%d is the largest number.", n2);
else
printf("%d is the largest number.", n3);
}

return 0;
}
```

Switch case: A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case. The switch statement allows us to execute one code block among many alternatives.

Syntax:

```
switch (expression)
{
case constant1:
// statements
break;

case constant2:
// statements
break;//optional
.
.
.
default:
```

Q. C program to calculate the weekday name by entering numbers

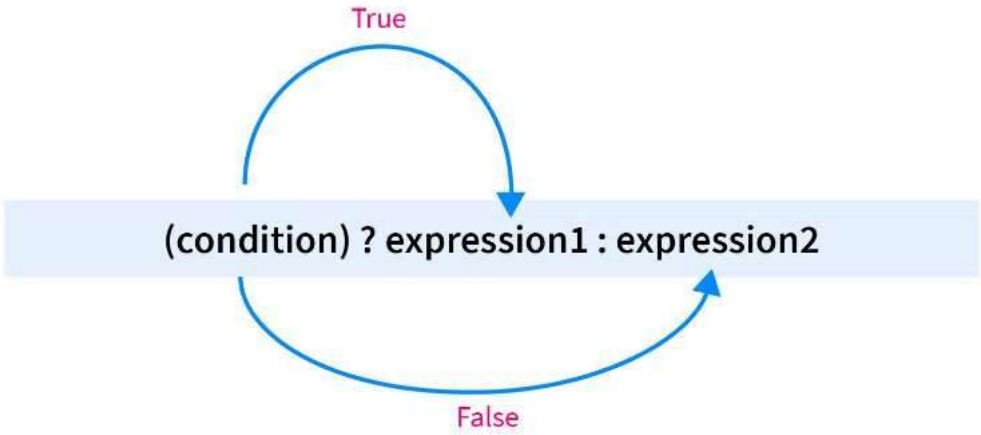
```
#include <stdio.h>

int main() {
int day = 4;

switch (day) {
case 1:
printf("Monday");
break;
case 2:
printf("Tuesday");
break;
case 3:
printf("Wednesday");
```

```
// default statements
}
```

```
break;
case 4:
printf(“Thursday”);
break;
case 5:
printf(“Friday”);
break;
case 6:
printf(“Saturday”);
break;
case 7:
printf(“Sunday”);
break;
}

return 0;
}
```

Conditional operator statement: It is a ternary operator and is used to perform simple conditional operations. It is used to do operations similar to if-else statement.

The general syntax of conditional operator is as under:-
Test expression? expression1:expression2

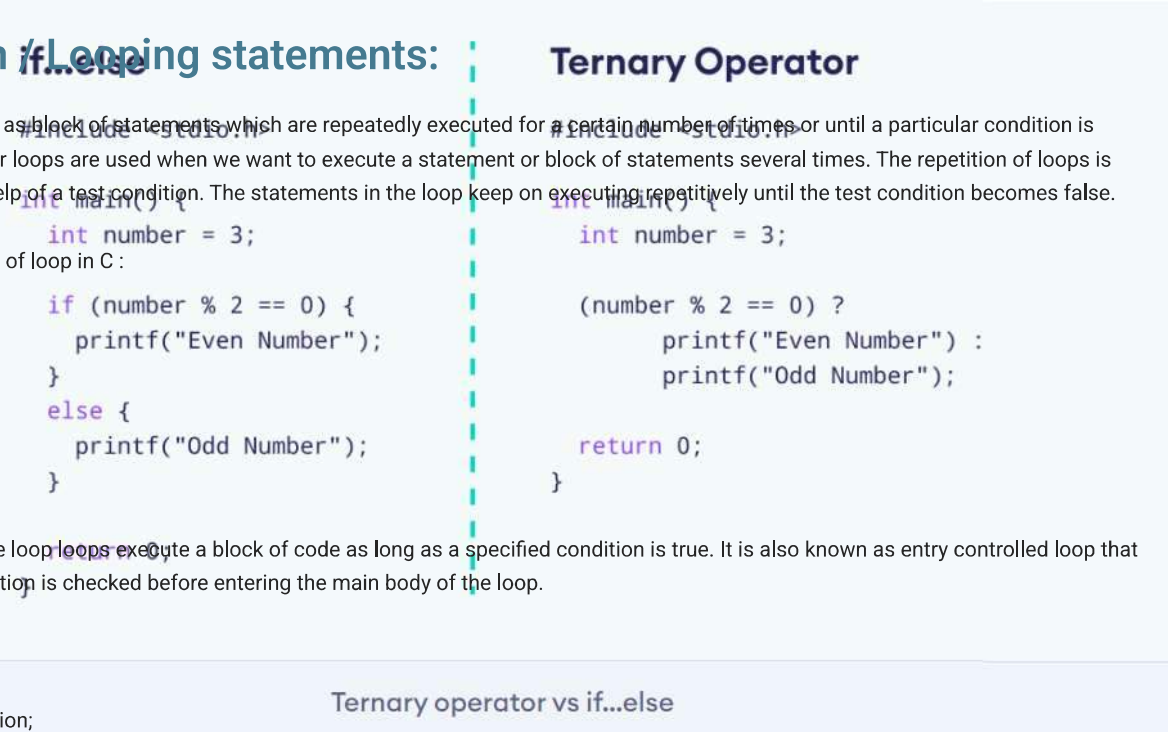


Example

Write a c program to demonstrate the conditional operator statement?

```
#include <stdio.h>

int main() {
int num;
scanf(“%d”, &num);
(num % 2 == 0)? printf(“The given number is even”) : printf(“The given number is odd”);

return 0;
}
```

2. Iteration / Looping statements:



Loop may be defined as block of statements which are repeatedly executed for a certain number of times or until a particular condition is satisfied. Iterations or loops are used when we want to execute a statement or block of statements several times. The repetition of loops is controlled with the help of a test condition. The statements in the loop keep on executing repetitively until the test condition becomes false.

There are three types of loop in C :

1. While loop
2. Do-while loop
3. For loop

While loop: The while loop loops execute a block of code as long as a specified condition is true. It is also known as entry controlled loop that means the test condition is checked before entering the main body of the loop.

Syntax:

```
initialization_expression;
while (test_expression)
{
// body of the while loop
update_expression;
}
```

Questions for students:

1. **Write a c program to find greatest number among two numbers using conditional operator statement.**
2. **Write a c program to print candidate can vote if candidate age is greater or equal to 18 and print candidate cannot vote if not greater than equal to age of 18.**

Example:

```
#include <stdio.h>

int main () {

/* local variable definition */
int a = 10;

/* while loop execution */
while( a < 20 ) {
printf("value of a: %d\n", a);
a++;
}
```

```
return 0;
}
```

Do-while loop: The do-while loop is similar to a while loop but the only difference lies in the do-while loop test condition which is tested at the end of the body. In the do-while loop, the loop body will execute at least once irrespective of the test condition. In case of do-while, firstly the statements inside the loop body are executed and then the condition is evaluated. As a result of which this loop is executed at least once even if the condition is initially false. After that the loop is repeated until the condition evaluates to false. Since in this loop the condition is tested after the execution of the loop, it is also known as posttest loop. It is also called exit controlled loop means the means the test condition is evaluated at the end of the loop body.

Syntax:

```
initialization_expression;
do
{
// body of do-while loop
update_expression;

} while (test_expression);
```

Example of Do while loop:

```
#include <stdio.h>

int main() {
int i = 0;

do {
printf("%d\n",i);
i++;
}while(i<=5);
return 0;
}
```

Difference between while and do-while loop

While	do-while
While loop is pre test loop.	do-while is post test loop
The statements of while loop may not be executed even once	The statements of do-while loop are executed atleast once
There is no semi colon given after while(condition)	There is semi colon given after while(condition);

The syntax of while loop is
While(condition)
{
Statements
}
	The syntax of do-while loop is as under:-
	Do
	{
	Statements;
	}while(condition);

For loop: The for loop allows to execute block of statements for a number of times. When number of repetitions is known in advance, the use of for loop will be more efficient. Thus this loop is also known as a determinate or definite loop. For loop consists of three expressions with semicolons.

for(initialization; test_condition; increment or decrement) { Statements or body of loop; }

Example: Write a program to write a table of n number given by user.

```
#include<stdio.h>
int main(){
int i=1,number;
printf("Enter a number: ");
scanf("%d",&number);
for(i=1;i<=10;i++){
printf("%d \n",(number*i));
}
return 0;
}
```

```
/* C program to calculate a Factorial of a given number */
#include <stdio.h>

int main()
{
int num,i;
long fact=1;
printf("Enter number");
scanf("%d",&num);
for(i=1;i<=num;i++)
fact=fact*i;
printf("%ld",fact);

return 0;
}
```

C program to generate Fibonaci series (most important, will apear in exam question)

```
#include <stdio.h>
int main() {

int i, n;

// initialize first and second terms
int t1 = 0, t2 = 1;

// initialize the next term (3rd term)
int nextTerm = t1 + t2;

// get no. of terms from user
printf("Enter the number of terms: ");
scanf("%d", &n);

// print the first two terms t1 and t2
printf("Fibonacci Series: %d, %d, ", t1, t2);

// print 3rd to nth terms
for (i = 3; i <= n; ++i) {
printf("%d, ", nextTerm);
t1 = t2;
t2 = nextTerm;
nextTerm = t1 + t2;
}
```

```
return 0;
}
```

Nested Loop

Nested loop: Using a loop inside another loop is called nested loop. C support n times of nested loop. The nested for loop means any type of loop which is defined inside the ‘for’ loop.

Syntax of Nested loop

```
Outer_loop
{
Inner_loop
{
// inner loop statements.
}
// outer loop statements.
}
```

Example of nested loop to print pattern:

```
*
**
***
****
*****
```

```
#include <stdio.h>

int main() {
int i, j, n;
// Ask the user for the number of rows
printf(“Enter the number of rows: “);
scanf(“%d”, &n);

// Outer loop for rows
for (i = 1; i <= n; i++) {
// Inner loop for columns (stars in each row)
for (j = 1; j <= i; j++) {
printf(“*”);
}
// Move to the next line after each row
printf(“\n”);
```

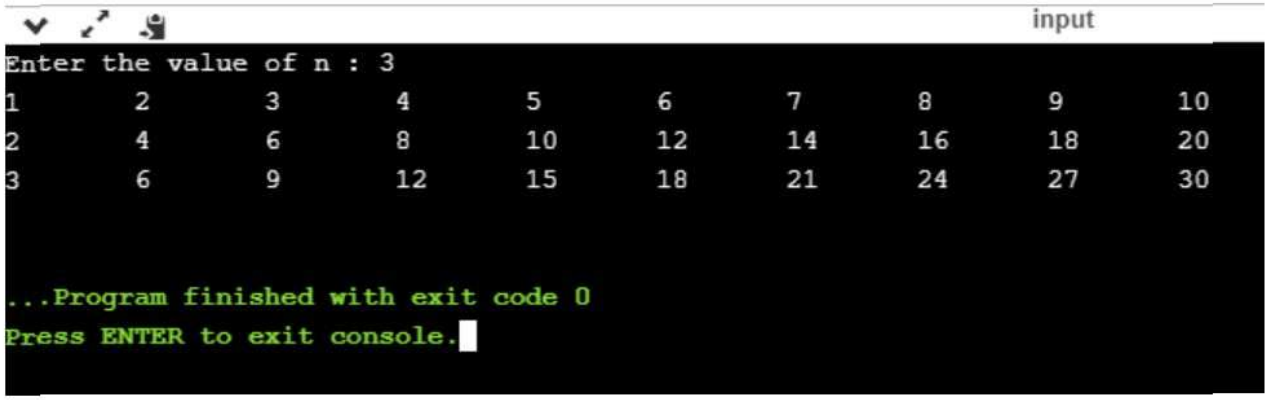
```
}
return 0;
}
```

2. Example of nested for loop

```
#include <stdio.h>
int main()
{
int n;// variable declaration
printf("Enter the value of n :");

scanf("%d",&n);

// Displaying the n tables.
for(int i=1;i<=n;i++) // outer loop
{
for(int j=1;j<=10;j++) // inner loop
{
printf("%d\t",(i*j)); // printing the value.
}
printf("\n");
}
```

Explanation of the above code

- First, the 'i' variable is initialized to 1 and then program control passes to the i<=n.
- The program control checks whether the condition 'i<=n' is true or not.
- If the condition is true, then the program control passes to the inner loop.
- The inner loop will get executed until the condition is true.
- After the execution of the inner loop, the control moves back to the update of the outer loop, i.e., i++.
- After incrementing the value of the loop counter, the condition is checked again, i.e., i<=n.
- If the condition is true, then the inner loop will be executed again.
- This process will continue until the condition of the outer loop is true.



3. Jump Statements / Loop Interrupts

In C, you can control the flow of loops using **loop interrupts** such as `break`, `continue`, and `return`. These keywords allow you to alter the normal flow of control within loops and functions.

Jump statements in C are used to alter the normal sequence of execution of a program. They allow the program to transfer control to a different part of the code.

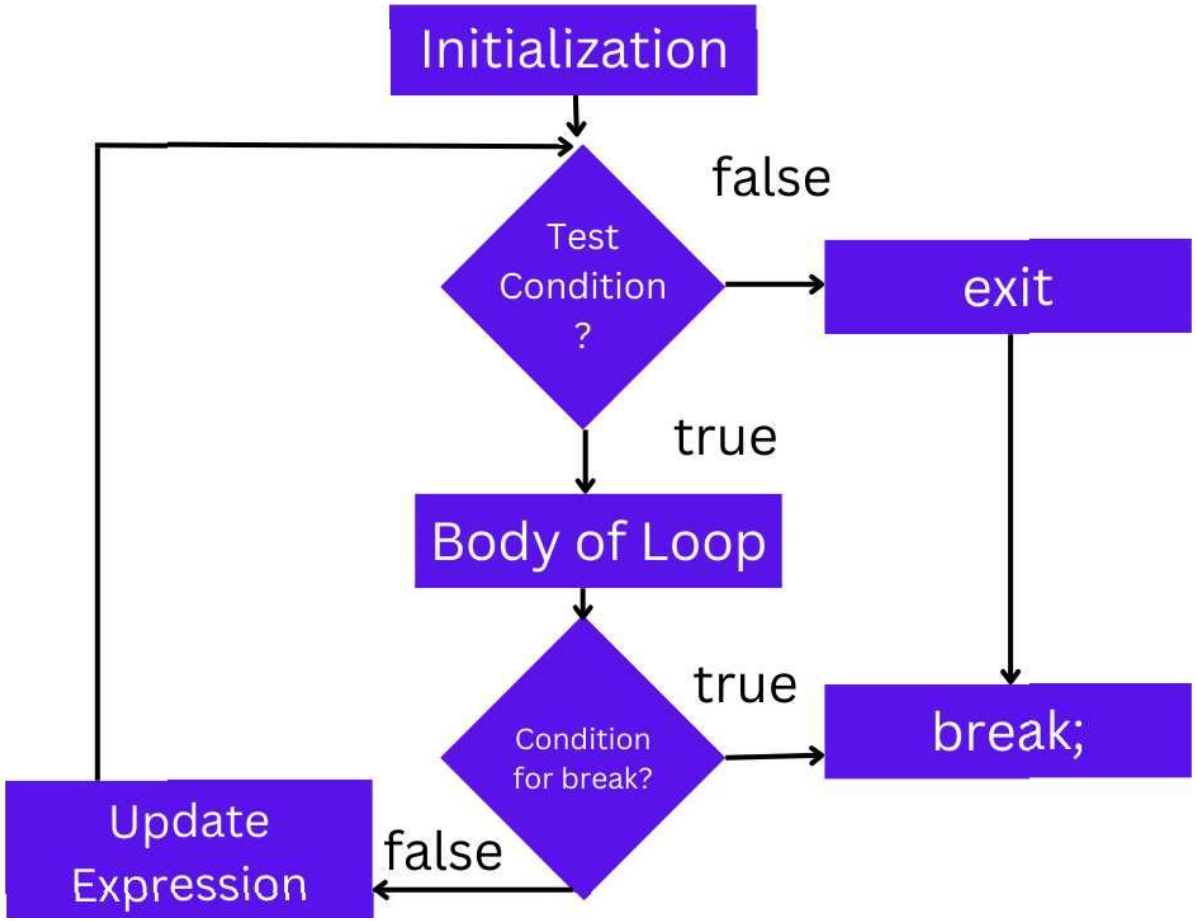
Simple definition: Jump statements are used to transfer the control from one part of the program to another part.

Types of Jump Statements in C
The break Statement terminates the execution of the loop and the control transferred to the statement immediately following the loop.
There are four types of jump statements:

Break statement is used inside the loops or switch statement. This statement causes an immediate exit from the loop or the switch case block in which it appears. If the test condition is to be terminated instantly without testing termination condition, the break statement is useful.

1. break
2. continue
3. goto
4. return

It can be written as
break;



Flowchart of break statement with for loop

Simple Progam example of break statement:

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

int main()
{
int i;
for(i=1; i<10; i++)
{
printf("%d ",i);
if(i==5)
break;
```

Continue statement

The continue statement skips the current iteration of the loop and continues with the next iteration. The continue statement is used inside the body of loop statement. it is used when we want to go to the next iteration of the loop after skipping the some of the statement of loop.

```
}
```

The difference between break and continue is that when a break statement is encountered the loop terminates and the control is transferred to the next statement following the loop, but when a continue statement is encountered the loop is not terminated and the control is transferred to the beginning of the loop.

Its syntax is:

```
continue;
```

Program example using continue:

```
#include<stdio.h>
int main()
{
int i;
for(i=1;i<=10;i++){
if(i==5){
continue;
}
printf("%d\n",i);
}
printf("out of for loop");
return 0;

}
```

output:

```
1
2
3
4
6
7
8
8
10
out of for loop
```

goto statement

Goto statement in C is a jump statement that is used to jump from one part of the code to any other part of the code in C. Goto statement helps in altering the normal flow of the program according to our needs. This is achieved by using labels, which means defining a block of code with a name, so that we can use the goto statement to jump to that label.

Syntax:

```
label_name:
.statement1;
.statement2;
.
.statementn;
goto label_name;
```

Q.Simple program using goto statement

```
#include<stdio.h>
void main()
{
int a=1;
repeat:
  if(a<=10)
  printf(" %d",a);
  a++;
goto repeat;
}
```

Output:
12345678910

Function

A function is a block of code that performs a specific task when it is called.
The function is also known as procedureor subroutine in other programming languages. Function enable us to write code separately for different functions.

Syntax:

```
return_type function_name( parameter list )
{
body of the function
}
```

Types of function
There are two types of function in C programming:

1. Standard library functions
2. User-defined functions

Standard library functions: Library functions are predefined / built in functions.
For example: printf(), scanf();, getch(), sqrt(), etc. These functions are defined in header files.

User Defined functions: Those functions that are created by user as per his/her need. Such Functions are known as User Defined functions.

Function prototype, Definition, and call

- Function prototype is also called funciton declaration. Function declaration is done above of the main function. Passing arguments in the function declaration and function definition are formal parameters which copied the values from actual parameters. Formal parameters are always in variables.
- Function Definition is a block of code that define the specific task that are enclosed with { }.
- Function call is done inside the main function. Passing the arguments while calling function is actual parameters. Actual parameters can be both values or variables.

Define actual and formal parameters?
Actual and formal parameters are two different forms of parameters that we use while declaring, defining and invoking a function. The actual parameter is the one that we pass to a function when we invoke it. On the other hand, a formal parameter is one that we pass to a function when we declare and define it.

BASIS FOR COMPARISON	ACTUAL PARAMETER	FORMAL PARAMETER
Definition	They are actual values passed to a function on which the function will perform operations	They are the variables in the function definition that would receive the values when the function is invoked
Occurrence	It occurs when we invoke a function	It occurs when we declare and define a function
Provided by	Either by the programmer or by the user	Only by programmer
Data types	Data types are not mentioned with the actual parameters	Data types are mentioned along the formal parameters
Form	It can be a value or a variable	It is always a variable

Example	add (a, b);	int add (int x, int y) {
		//body
		}

Different Ways of Calling a Function:

Depending on whether the function accepts arguments or not and returns a value or not, there can be four different aspects of C function calls, which are:

```
// Example of Function Without arguments and without return

#include<stdio.h>
void greater();

void greater()
{
int a,b;
   printf(“enter two numbers:”);
   scanf(“%d%d”,&a,&b);
if(a>b)
{
printf(“%d is greater”, a);
}
else
{
printf(“%d is greater”, b);
}
}
int main(){
greater();
return 0;

}
```

```
//Example of function with arugments and with return

#include<stdio.h>
int greater();
int greater( int a, int b)
{
if(a>b)
return a;
else
return b;
}
int main()
{
int a,b,c;
printf(“enter a and b:”);
scanf(“%d%d”,&a,&b);
c=greater(a,b); //
printf(“greater is %d”,c);
return 0;
}
```

```
//Example of Function with argument without return

#include<stdio.h>
void greater(int x, int y);

void greater(int a, int b) // formal parameters
{
if(a>b)
{
printf(“%d is greater”, a);
}
else
{
printf(“%d is greater”, b);
}
}
int main(){
   int x,y;
   printf(“enter two numbers:”);
   scanf(“%d%d”,&x,&y);
greater(x,y); // actual parameters are copied to formal parameter
return 0;

}
```

```
// Example of Function Without arguments and with return

#include<stdio.h>
int greaternumber();

int greaternumber(){
int a,b;
printf(“enter two numbers:\n”);
scanf(“%d%d”,&a,&b);
if(a>b)
return a;
else
return b;
}
int main()
{
int greater;
greater=greaternumber();
printf(“%d is greater”,greater);
return 0;

}
```

TO perferm sum

```
// Example of Function Without arguments and without return

#include<stdio.h>
void sum();

void sum()
{
int a,b, c;
```

To Perform sum

```
//Example of function with arugments and with return

#include<stdio.h>
int sum(int int);
int sum(int x, int y)
{
int z=x+y;
return z;
```

To Perform Sum

```
//Example of Function with argument without return

#include<stdio.h>
void sum(int int);
void sum(int x, int y)
{
int s;
s=x+y;
printf(“sum is %d”,s);
```

To perform sum

```
// Example of Function Without arguments and with return

#include<stdio.h>
int sum();

int sum(){
int x,y, s;
printf(“enter two numbers:\n”);
```

```
intf(“enter two numbers:”);
anf(“%d%d”,&a,&b);
c=a+b;
printf(“Sum is %d”, c);
int main(){
sum();
return 0;

}
```

```
}
int main()
{
int a,b,s;
printf(“enter two number\n:”);
scanf(“%d%d”, &a, &b);

s=sum(a,b);

prinftf(“%d is sum”, s);
return o;
}
```

```
}
int main()
{
int a,b,s;
printf(“enter two number\n:”);
scanf(“%d%d”, &a, &b);

s=sum(a,b);
return o;
}
```

```
scanf(“%d%d”,&a,&b);
s=x+y;
return s;
}
int main()
{
int a;
a=sum();
printf(“%d is sum”,a);
return 0;

}
```

There are two methods to pass the data into the function in C language, i.e., call by value and call by reference.

HOW TO CALL C FUNCTIONS IN A PROGRAM?

1. Call by value
2. Call by reference

1. CALL BY VALUE:

In call by value method, the value of the variable is passed to the function as parameter.

The value of the actual parameter can not be modified by formal parameter.

Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

Actual parameter – This is the argument which is used in function call.

Formal parameter – This is the argument which is used in function definition

EXAMPLE PROGRAM FOR C FUNCTION (USING CALL BY VALUE):

In this program, the values of the variables “m” and “n” are passed to the function “swap”.

These values are copied to formal parameters “a” and “b” in swap function and used.

```
#include<stdio.h>
// function prototype, also called function declaration
void swap(int a, int b);
int main()
{
int m = 22, n = 44;
// calling swap function by value
printf(” values before swap m = %d \nand n = %d”, m, n);
swap(m, n);
}
void swap(int a, int b)
{
int tmp;
tmp = a;
```

```
a = b;
b = tmp;
printf(" \nvalues after swap m = %d\n and n = %d", a, b);
}
```

2. CALL BY REFERENCE:

In call by reference method, the address of the variable is passed to the function as parameter.
The value of the actual parameter can be modified by formal parameter.
Same memory is used for both actual and formal parameters since only address is used by both parameters.
EXAMPLE PROGRAM FOR C FUNCTION (USING CALL BY REFERENCE):
In this program, the address of the variables "m" and "n" are passed to the function "swap".
These values are not copied to formal parameters "a" and "b" in swap function.
Because, they are just holding the address of those variables.
This address is used to access and change the values of the variables.

```
#include<stdio.h>
// function prototype, also called function declaration
void swap(int *a, int *b);
int main()
{
int m = 22, n = 44;
// calling swap function by reference
printf("values before swap m = %d \n and n = %d",m,n);
swap(&m, &n);
}
void swap(int *a, int *b)
{
int tmp;
tmp = *a;
*a = *b;
*b = tmp;
printf("\n values after swap a = %d \nand b = %d", *a, *b);
}
```

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.

3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

swapping.c

```c
#include<stdio.h>
void swap(int, int);
void swap(int x, int y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
    printf("values after swapping are %d,%d",x,y);
}
int main(){
    int a,b;
    printf("enter two numbers for swapping:\n");
    scanf("%d%d",&a,&b);
    printf("values before swap are: %d,%d\n",a,b);
    swap(a,b);
    return 0;
}
```

```
C:\Users\Asus\Desktop\C pro
enter two numbers for swapping:
10
30
values before swap are: 10,30
values after swapping are 30,10
--------------------------------
Process exited after 5.003 seconds with return value 0
Press any key to continue . . .
```

swapping.c

```c
#include<stdio.h>
void swap(int *x, int*y);
void swap(int *x, int *y)
{
    int temp;
    temp=*x;
    *x=*y;
    *y=temp;
    printf("values after swapping are %d,%d",*x,*y);
}
int main(){
    int a,b;
    printf("enter two numbers for swapping:\n");
    scanf("%d%d",&a,&b);
    printf("values before swap are: %d,%d\n",a,b);
    swap(&a,&b);
    return 0;
}
```

```
C:\Users\Asus\Desktop\C pro
enter two numbers for swapping:
10
50
values before swap are: 10,50
values after swapping are 50,10
--------------------------------
Process exited after 6.964 seconds with return value 0
Press any key to continue . . .
```

Recursion

Function that call itself is called recursion. While using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinte loop.

Syntax:

```
void recursion() {
recursion(); /* function calls itself */
}
int main()
{
recursion();
}
```

//Example of recursion program with argument with return.

```
#include<stdio.h>
int factorial(int x); //function declaration with argument
int main() //main function
```

//Example of recursion function to find n terms of ficonacci series

```
#include <stdio.h>
int fibonacci(int); // function declaration
```

```
{
int a; //declaration for input n number to find its factorial
printf("enter number to find factorial:");
scanf("%d",&a); // taking user input number
int fn; //assume variable to store function call, you also can call it directly.
fn=factorial(a); //function call by value with actual argument
printf("factorial is %d", fn);  //displaying function call value
return 0;
}
factorial(x)//function defintion with formal argument
{
if(x==1)
return 1;
else
return x*factorial(x-1); //function call itself
}
```

```
//function definition section start
int fibonacci(int x) {

if(x == 0) {
return 0;
}
if(x == 1) {
return 1;
}
else
return fibonacci(x-1) + fibonacci(x-2); //function call itself
}

//main function section start
int main() {
int a;
printf("enter a number:");
scanf("%d",&a);
int y;
for (y = 0; y < a; y++) {
printf("%d\n", fibonacci(y));
}
return 0;
}
```

Unit-5 Array, Pointer String

Array:

An array is a collection of values of similar kinds of data types. Values in array accessed using array name with subscripts in brackets[]. Syntax of array declaration is:

```
data_type array_name[size];
```

```
#include<stdio.h>
int main()
{
int i,a[10]={10,20,39,58,19};
for(i=0;i<10;i++)
{
printf("%d\n",a[i]);
```

```
}
return 0;
}
```

Output:

```
10
20
39
58
19
0
0
0
0
0
```

Declaration and initialization of array

An array is a variable that can store multiple values. For example, if you want to store 100 integers, you can create an array for it.

```
dataTypearrayName[arraySize];
int data[100];
```

It is possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

You can also initialize an array like this.

```
int mark[] = {19, 10, 8, 17, 9};
```

Accessing array

Array can be accessed using array-name and subscript variable written inside pair of square brackets [].

for example:

```
arr[3] = Third Element of Array
arr[5] = Fifth Element of Array
arr[8] = Eighth Element of Array
```

```
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array

#include <stdio.h>
int main()
{
int i, array[5];
printf(“Enter 5 integers: “);
// taking input and storing it in an array
```

```
for(i = 0; i < 5; i++)
{
scanf(“%d”, &array[i]);
}
printf(“printing those values: “);
// printing elements of an array
for( i = 0; i < 5; ++i)
{
printf(“%d\n”, array[i]);
}
return 0;
}
```

```
//C Program to find smallest element in an array
#include<stdio.h>
int main() {
int a[30], i, num, smallest;
printf(“\nEnter no of elements :”);
scanf(“%d”, &num); //Read n elements in an array
for (i = 0; i < num; i++)
scanf(“%d”, &a[i]); //Consider first element as smallest
smallest = a[0];
for (i = 0; i < num; i++) {
if (a[i] < smallest) {
smallest = a[i]; } } // Print out the Result
printf(“\nSmallest Element : %d”, smallest);
return (0);
}
```

Multidimensional Arrays: An array with more than one index is called a multidimensional array. Syntax to declare a multidimensional array:

```
data_type array_name[][][]...;
```

write a c program to add 2*2 two matrices ?

```c
#include <stdio.h>
int main() {
    // Declaration and initialization of the first matrix
    int matrix1[2][2] = {{1, 2},
                         {3, 4}};

    // Declaration and initialization of the second matrix
    int matrix2[2][2] = {{5, 6},
                         {7, 8}};

    // Declaration of the resultant matrix
    int result[2][2];
    int i,j;

    // Adding corresponding elements of matrix1 and matrix2
    for( i = 0; i < 2; i++) {
        for( j = 0; j < 2; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    // Displaying the result
    printf("Resultant Matrix:\n");
    for( i = 0; i < 2; i++) {
        for( j = 0; j < 2; j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

```
C:\Users\Asus\Desktop\add2.

Resultant Matrix:
6 8
10 12

------------------------
Process exited after 0.031
Press any key to continue
```

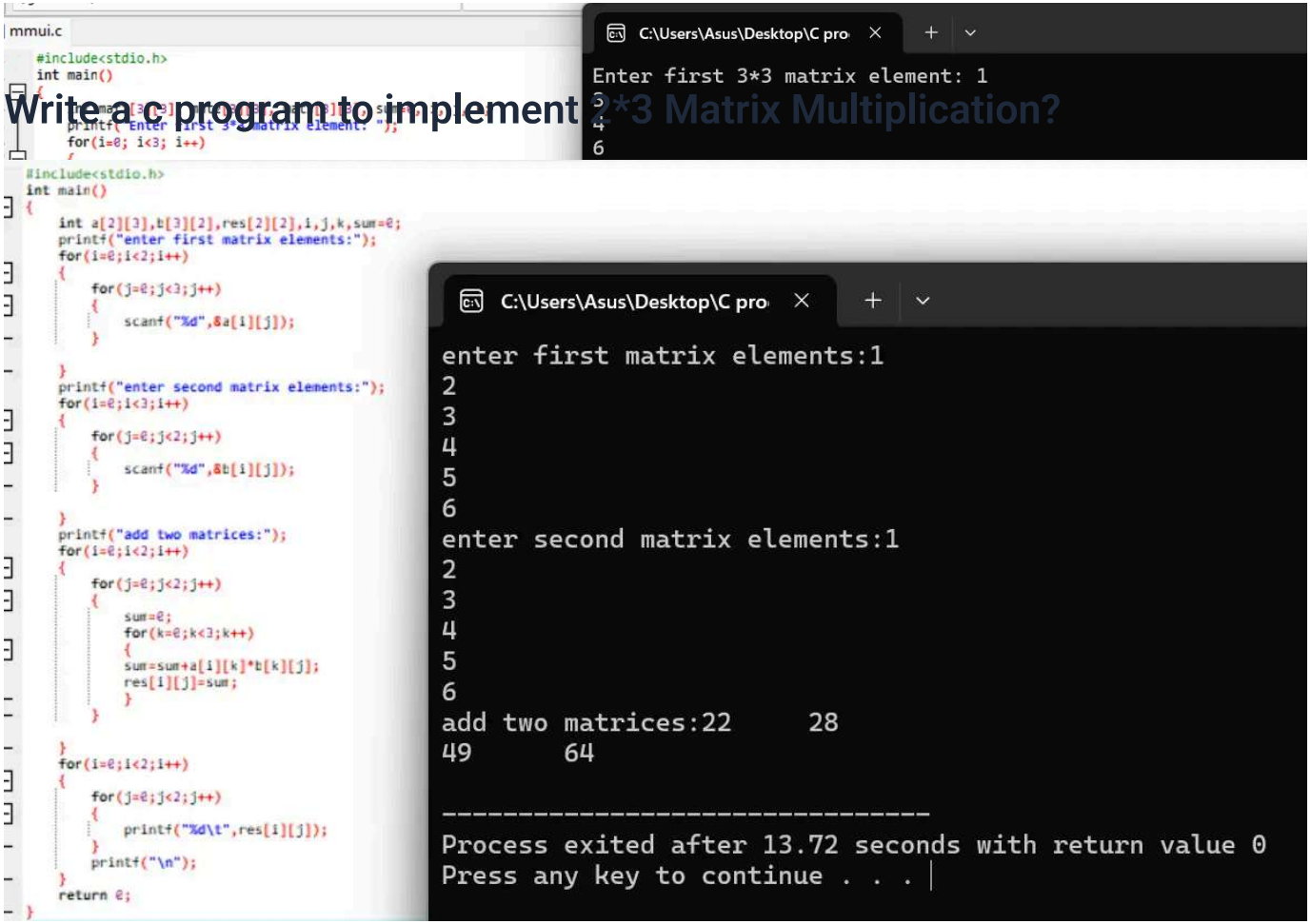
C program to add two 2x2 matrices based on user input ?

```c
#include<stdio.h>
int main()
{
    int matrix1[2][2], matrix2[2][2], result[2][2],i,j;
    printf("enter the values of first matrix:");
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            scanf("%d", &matrix1[i][j]);
        }
    }
    printf("enter the values of second matrix:");
        for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            scanf("%d",&matrix2[i][j]);
        }
    }
    printf("add two matrices:");
        for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            result[i][j]=matrix1[i][j]+matrix2[i][j];
            printf("%d\t",result[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

```
C:\Users\Asus\Desktop\addm
enter the values of first matrix:1
2
3
4
enter the values of second matrix:5
6
7
8
add two matrices:6      8
10      12

--------------------------------
Process exited after 6.093 seconds with return value 0
Press any key to continue . . .
```

c program to multiply 3*3 matrices by taking matrix values from user input.

Write a c program to implement 2*3 Matrix Multiplication?



```c
// c program to multiply m1[3][4], m2[4][2]
#include<stdio.h>
int main()
{
        int res[3][2],m1[3][4],m2[4][2],i,j,k,sum=0;
        printf("enter the values of first matrix:");
        for(i=0;i<3;i++)
        {
                for(j=0;j<4;j++)
                {
                        scanf("%d",&m1[i][j]);
                }
        }
        printf("enter the values of second matrix:");
        for(i=0;i<4;i++)
        {
                for(j=0;j<2;j++)
                {
                        scanf("%d",&m2[i][j]);
                }
        }
        printf("performing multiplication");
        for(i=0;i<3;i++)
        {
                for(j=0;j<2;j++)
                {
                        sum=0;
                        for(k=0;k<4;k++)
                        {
                                sum=sum+m1[i][k]*m2[k][j];
                                res[i][j]=sum;
                        }
                }
```

```
}
printf("displaying result of multiplication");
for(i=0;i<3;i++)
}
```

Difference Between Malloc and Calloc Memory Allocation.

Malloc Memory Allocation	Calloc Memory Allocation
Malloc Stands for memory allocation.	Calloc Stands for contiguous memory allocation.
Malloc creates a single memory block of a user-specified size.	Calloc can allocate multiple memory blocks to a variable.
The malloc function is initialized to garbage values if no value given.	The calloc function are always initialized to Zero.
Malloc is faster in speed.	Calloc is slower than malloc in speed.
The number of arguments is 1 i.e byte_size.	The number of arguments is 2 i.e number of memory blocks and size of memory block.
Syntax: int*ptr=(int*)malloc(n*sizeof(int));	**Syntax:** int*ptr=(int*)calloc(n,sizeof(int));

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

int main() {
int *num1, *num2, *sum;

// Allocating memory for integers using malloc
num1 = (int*) malloc(sizeof(int));
num2 = (int*) malloc(sizeof(int));
sum = (int*) malloc(sizeof(int));

if (num1 == NULL || num2 == NULL || sum == NULL) {
printf("Memory allocation failed!\n");
return 1;
}

// Input numbers
printf("Enter first number: ");
scanf("%d", num1);
printf("Enter second number: ");
scanf("%d", num2);
```

```
// Perform addition
*sum = *num1 + *num2;

// Display result
printf("Sum: %d\n", *sum);

// Free allocated memory
free(num1);
free(num2);
free(sum);

return 0;
}
```

Concept of Pointer, pointer address, dereference, declaration,assignment, intializaion

Pointer

A pointer is a variable that stores the memory address of another variable as its value. Pointer variable is always preceded by * operator.
if a pointer variable p is declared as : int *p;
it signifies p is pointer variable and it can store address of integer variable (i.e. it can not store address of other type's variables.
* operator is deference operator
& operator is reference operator

An indirection operator, is an operator used to obtain the value of a variable to which a pointer points. While a pointer pointing to a variable provides an indirect access to the value of the variable stored in its memory address, the indirection operator dereferences the pointer and returns the value of the variable at that memory location. The indirection operator is a unary operator represented by the symbol (*). The indirection operator is also known as the dereference operator.

Valid Examples:

```
int *p;
int num;
p=&num;
```

Invalid Examples:

```
int *p;
float num;
p=&num; /* invalid pointer variable p cannot store address of float variable */
```

Pointer Declaration

A pointer variable is declared as follows:

Syntax:
data_type * variable_name;

here, *is called indirection or dereference operator and variable_name is now pointer.

Example:
int *x;
float *y;
char *rajesh;

Simple Program Example:
#include<stdio.h>
int main()
{
int a=50;
int* b=&a;
printf("%d is value of a variable\n",a);
printf("%d is the value of pointer\n",*b);

printf("%p is the memory address of pointer variable\n", b);
printf("%p is the address of pointer\n",&a);

return 0;
}
Output:
50 is value of a variable
50 is the value of pointer
000000000062FE14 is the memory address of pointer variable
000000000062FE14 is the address of pointer

Write a c program to add to numbers using Pointer

pointer_sum.c

```c
#include<stdio.h>
int main(){
    int a,b,sum;
    printf("enter two numbers:\n");
    scanf("%d %d",&a, &b);
    int *x=&a;
    int *y=&b;
    sum=*x+*y;
    printf("sum is %d",sum);
    return 0;
}
```

```
C:\Users\Asus\Desktop\C pro
enter two numbers:
30
50
sum is 80
--------------------------------
Process exited after 6.852 seconds with return value
Press any key to continue . . .
```

Pointer Arithmetic

Pointer Arithmetic refers to arithmetic operations on pointers. Since pointers store memory addresses, Pointer Arithmetic operations help navigate through memory locations efficiently.

1. Types of Pointer Arithmetic

You can perform the following operations with pointers in C:

1. **Increment** (`ptr++`) – Moves to the next memory location.
2. **Decrement** (`ptr--`) – Moves to the previous memory location.
3. **Addition** (`ptr + n`) – Moves `n` positions forward.
4. **Subtraction** (`ptr - n`) – Moves `n` positions backward.
5. **Pointer difference** (`ptr2 - ptr1`) – Finds the number of elements between two pointers.

1:Pointer Increment & Decrement

```
#include <stdio.h>
int main() {
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr; // Pointer to the first element
printf("Initial pointer value: %d\n", *ptr);
ptr++; // Move to the next element
printf("After increment: %d\n", *ptr);
ptr--; // Move back to the first element
printf("After decrement: %d\n", *ptr);
return 0;
}
```

Output:
Initial pointer value: 10
After increment: 20
After decrement: 10

2: Pointer Addition and Subtraction

```
#include <stdio.h>
int main() {
int arr[] = {5, 10, 15, 20, 25};
int *ptr = arr; // Pointer to the first element
printf("Value at ptr: %d\n", *ptr);
ptr = ptr + 2; // Move two positions forward
printf("After ptr + 2: %d\n", *ptr);
ptr = ptr – 1; // Move one position backward
printf("After ptr – 1: %d\n", *ptr);
return 0;
}
```

```
#include <stdio.h>

int main() {
int num = 10;
int *ptr = &num; // Pointer pointing to num
printf("Address of num: %p\n", ptr);
ptr = ptr + 1; // Move the pointer forward by 1 int (4 bytes)
printf("Address after addition: %p\n", ptr);
ptr = ptr – 1; // Move the pointer back to original position
printf("Address after subtraction: %p\n", ptr);
return 0;
}
```

STRINGS:

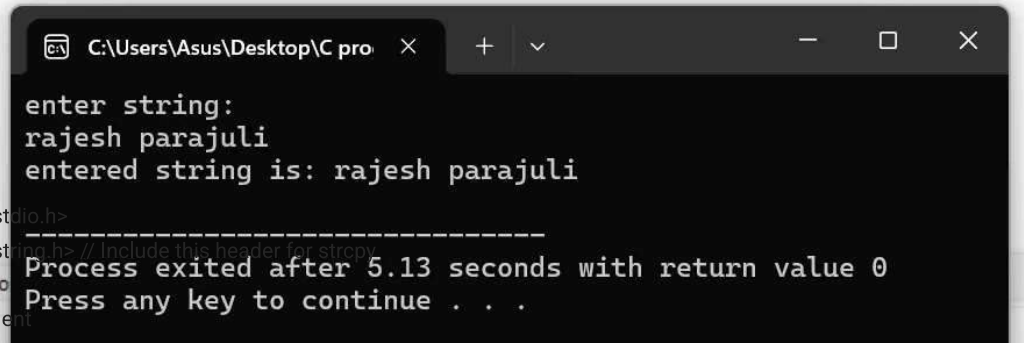
An array of characters are known as Strings. There are various built-in string handling functions in c. Some of them are:

1. strcpy()
2. strcat()
3. strcmp()

4. strcmpi()
5. strupr()
6. strlwr()
7. strrev()

```
#include <stdio.h>
#include <string.h> // Required for string handling functions

// Main function
int main() {
// Declare string variables
char str1[50] = “Hello”;
char str2[50] = ” World”;
char str3[50];

// strcpy: Copying string
strcpy(str3, str1); // Copy content of str1 into str3
printf(“After strcpy, str3: %s\n”, str3); // Output: Hello

// strcat: Concatenate strings
strcat(str1, str2); // Concatenate str2 to str1
printf(“After strcat, str1: %s\n”, str1); // Output: Hello World

// strlen: Find the length of a string
int len = strlen(str1); // Get the length of str1
printf(“Length of str1: %d\n”, len); // Output: 12

// strupr: Convert string to uppercase
strupr(str1); // Convert str1 to uppercase
printf(“After strupr, str1: %s\n”, str1); // Output: HELLO WORLD

// strlwr: Convert string to lowercase
strlwr(str1); // Convert str1 back to lowercase
printf(“After strlwr, str1: %s\n”, str1); // Output: hello world

// strrev: Reverse the string
strrev(str1); // Reverse str1
printf(“After strrev, str1: %s\n”, str1); // Output: dlrow olleh

return 0;
}
```

```c
#include<stdio.h>
int main()
{
    char str[20];
    printf("enter string:\n");
    gets(str); //read string with space by compiler as scanf function
    printf("entered string is: ");
    puts(str); //display string
return 0;
}
```




```c
#include<stdio.h>
#include<string.h> // Include this header for strcpy
struct student
{
int id;
char name[30];
char gender[30];
int age;
};
```

Structure:

The structure is the collection of different data types grouped under the same name using the struct keyword. It is also known as the user-defined data type that enables the programmer to store different data type records in the Structure. Furthermore, the collection of data elements inside the Structure is termed as the member.

```c
int main() {
struct student st;
st.id = 1;

// Use strcpy to assign string values to character arrays
strcpy(st.name, “rajesh”);
strcpy(st.gender, “male”);

st.age = 28;

// Print the struct values
printf(“id is %d\n”, st.id);
printf(“name is %s\n”, st.name);
printf(“gender is %s\n”, st.gender);
printf(“age is %d\n”, st.age);
```

```
return 0;
}
```

Write a c program using structure to input staff id,name, and the salary of 50 staffs. Display staff id, name and salary of those staff whose salary range from 25 thousands to 40 thousand.

```
#include<stdio.h>
struct staff{
int id;
char name[30];
double salary;
};
int main()
{
   struct staff s[50];
   int i;
   //input data of 50 students
   for(i=0;i<50;i++)
   {
   printf("\n%d staff details:  ",i+1);
   printf("\nenter staff id: ");
   scanf("%d", &s[i].id);
   printf("enter staff name: ");
   scanf("%s", s[i].name);
   printf("enter staff salary : ");
   scanf("%lf", &s[i].salary);
   }
   //printing 50 students entered details with given condition
   printf("\nStaff with salary between 25,000 and 40,000:\n");
   for (i = 0; i < 50; i++) {
      if (s[i].salary >= 25000 && s[i].salary <= 40000) {
      printf("\n%d staff details: ",i+1);
      printf("\nstaff id:%d",s[i].id);
      printf("\nstaff name:%s",s[i].name);
      printf("\nstaff salary:%lf",s[i].salary);

      }
}
return 0;
}
```

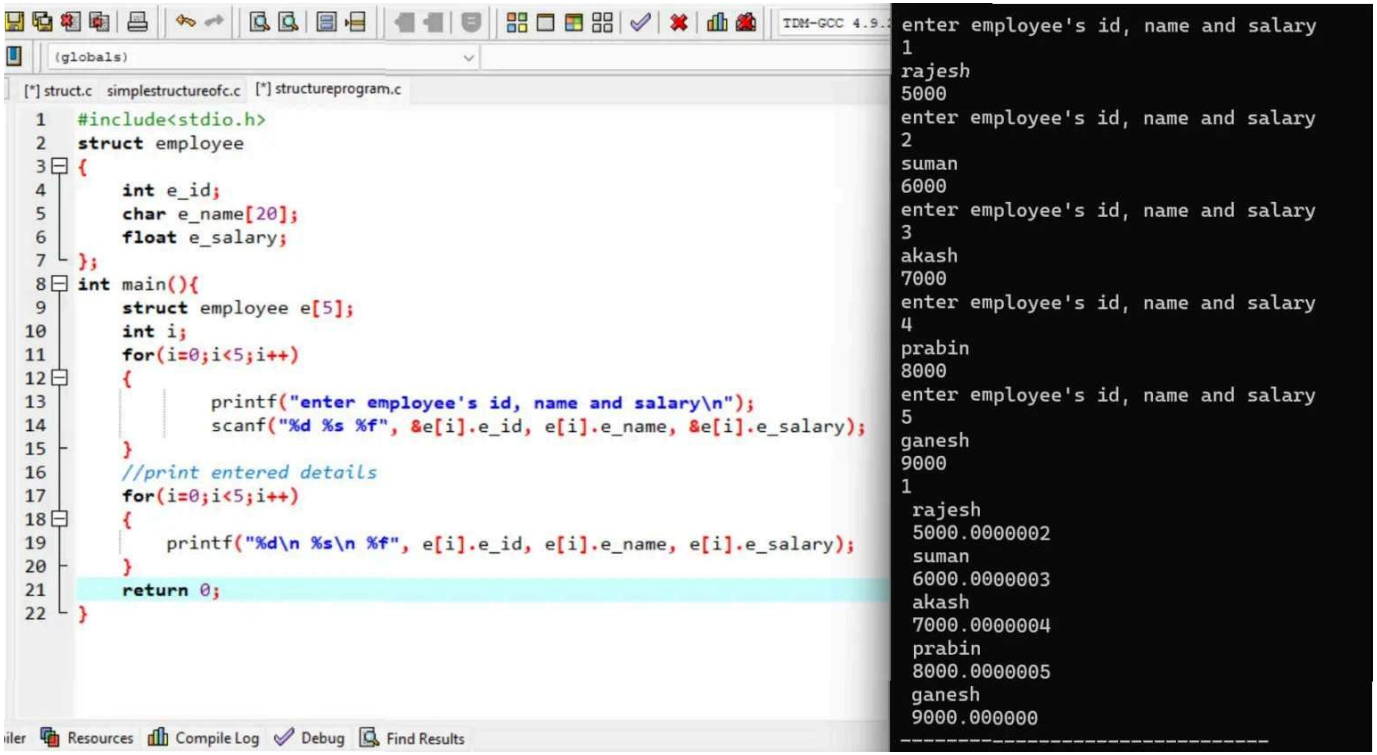
Simple c program using structure:

struct.c

```c
#include<stdio.h>
struct student
{
    int s_id;
    char s_name[20];
    char s_address[20];
    int s_marks;
};
int main(){
    struct student st;
    printf("enter student id:\n");
    scanf("%d",&st.s_id);
    printf("enter student name:\n");
    scanf("%s",st.s_name);
    printf("enter student adress:\n");
    scanf("%s",st.s_address);
    printf("enter student marks:\n");
    scanf("%d", &st.s_marks);
    printf("Displaying student information:\n");
    printf("%d is id\n", st.s_id);
    printf("%s is name\n", st.s_name);
    printf("%s is address\n", st.s_address);
    printf("%d is marks\n", st.s_marks);
    return 0;
}
```

```
C:\Users\Asus\Desktop
enter student id:
1
enter student name:
rajesh
enter student adress:
sukhad
enter student marks:
80
Displaying student information:
1 is id
rajesh is name
sukhad is address
80 is marks

--------------------------------
Process exited after 27.04 seconds with retur
n value 0
Press any key to continue . . .
```

Write a c program to store information of 5 employee(empid, name, salary) and display it using structure variable. (Most Important for Exam +2, BCA, BICTE)



Structure pointer

[*] sa.c structurepointer.c

```c
#include<stdio.h>
struct student
{
    int s_id;
    char s_name[20];
};
int main()
{
    struct student s;
    struct student *ptr = &s;
    printf("enter student id:\n");
    scanf("%d",&(*ptr).s_id);
    printf("enter student name:\n");
    scanf("%s",(*ptr).s_name);
    printf("the student id is %d\n", (*ptr).s_id);
    printf("the student name is %s", ptr->s_name);
    return 0;
}
```

```
C:\Users\Asus\Desktop\C pro
enter student id:
5
enter student name:
rajesh
the student id is 5
the student name is rajesh
--------------------------------
Process exited after 6.859 seconds with return value 0
Press any key to continue . . .
```

File Hanndling in C

A file is a collection of sequence of bytes(data) in one unit on the disk permanently. In general, file is a collection of related records such as student's name, date of birth, address, mobile phone number etc. Various operations can be done such as creating a file, opening a file, reading, writing, moving to a specific location in a file, and closing a file.

File handling is the process of storing data in the form of input or output produced by running C programs in data file for future reference and analysis. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.
File handling concept provides various operations like creating a file, opening a file, reading a file or manipulating data inside a file etc.

Why file handling?

The data stored in the various of a program will be lost once the program is terminated because they are stored in the Random Access Memory(RAM) which is volatile memory. So, if we want store that data(input/output) used in the program permanently inside the secondary storage device so that, we can access these data from there whenever it is needed file handling concept is important.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. Some operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

C provides a number of build-in function to perform basic file operations:

- `fopen()` - create a new file or open a existing file
- `fclose()` - close a file
- `getc()` - reads a character from a file
- `putc()` - writes a character to a file
- `fscanf()` - reads a set of data from a file
- `fprintf()` - writes a set of data to a file
- `getw()` - reads a integer from a file
- `putw()` - writes a integer to a file
- `fseek()` - set the position to desire point
- `ftell()` - gives current position in the file
- `rewind()` - set the position to the beginning point

Various File opening modes in c

- `r` - open a file in read mode
- `w` - opens or create a text file in write mode
- `a` - opens a file in append mode
- `r+` - opens a file in both read and write mode
- `a+` - opens a file in both read and write mode
- `w+` - opens a file in both read and write mode

File Operation Modes:

The different file mode that can be used in opening file are:

Mode	Description
"w" (write)	If the file doesn't exist then this mode creates a new file for writing, and if the file already exists then the previous data is erased and the new data entered is written to the file.
"r" (read)	This mode is used for opening an existing file for reading purpose only. The file to be opened must exist and the previous data of the file is not erased.
"a" (append)	If the file doesn't exist then this mode creates a new file and if the file already exists then the new data entered is appended at the end of existing data. In this mode, the data existing in the file is not erased as in "w" mode.
"w+" (write + read)	This mode is same as "w" mode but in this mode we can also read and modify the data. If the file doesn't exist then a new file is created and if the file exists then previous data is erased.
"r+" (read + write)	This mode is same as "r" mode but in this mode we can also write and modify existing data. The file to be opened must exist and the previous data of file is not erased. Since we can add new data and modify existing data so this mode is also called update mode.
"a+" (append + read)	This mode is same as the "a" mode but in this mode we can also read the data stored in the file. If the file doesn't exist, a new file is created and if the file already exists then new data is appended at the end of existing data. We cannot modify existing data in this mode

Creating a file in c

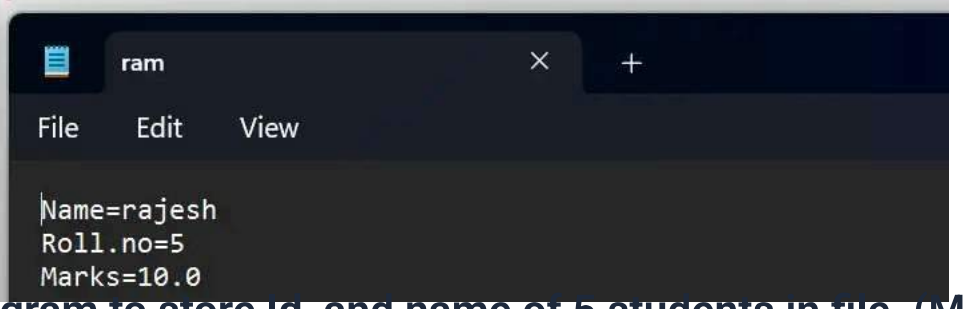
[*] createfileinc.c

```c
//c program to create a file
#include<stdio.h>
int main(){
    FILE *ptr=NULL; //declare file pointer and assigned it to NULL value because of not to store garbage value
    ptr=fopen("rajesh.txt","w"); //create a text file named rajesh
    fclose(ptr); //closing a file with providing pointer variable as argument
    return 0;
}

```

Write a c program to write name, roll no and marks of students in file using fprintf function.

filehandlingdatareading.c

```c
#include<stdio.h>
int main()
{
FILE *fp;
char name[20];
int rollno;
float marks;
fp=fopen("ram.txt","w");
if(fp==NULL)
{
printf("File can not open.");
}
printf("Enter Name: ");
gets(name);
printf("Enter roll.no: ");
scanf("%d",&rollno);
printf("Enter marks: ");
scanf("%f",&marks);
printf("Now writing data into file.....");
fprintf(fp,"Name=%s\nRoll.no=%d\nMarks=%.1f",name,rollno,marks);
fclose(fp);
getch();
return 0;
}
```



Write a c program to store id, and name of 5 students in file. (Most important for +2, BCA, BICTE, BIT)

```c
#include<stdio.h>
struct student
{
    int st_id;
    int st_name[20];
};
int main(){
    FILE *ptr;
    struct student s[5];
    int i;

    for(i=0;i<5;i++)
    {
    printf("enter id and name of students:");
    scanf("%d %s",&s[i].st_id,s[i].st_name);
    }
    //to write these information on our file
    ptr=fopen("studentfile.txt", "w");
    if(ptr==NULL)
    {
        printf("file not open");
    }
    for(i=0;i<5;i++)
    {
    fprintf(ptr,"Id=%d, Name=%s\n",s[i].st_id, s[i].st_name);//storing data into file
```

```
C:\Users\Asus\Desktop\C pro
enter id and name of students:1
rajesh
enter id and name of students:2
suman
enter id and name of students:3
akash
enter id and name of students:4
aashish
enter id and name of students:5
mahendra
Id=1, Name=rajesh
Id=2, Name=suman
Id=3, Name=akash
Id=4, Name=aashish
Id=5, Name=mahendra

--------------------------------
Process exited after 41.1 seconds with return value 0
Press any key to continue . . .
```

```
File    Edit    View

Id=1, Name=rajesh
Id=2, Name=suman
Id=3, Name=akash
Id=4, Name=aashish
Id=5, Name=mahendra
```

Formatted input output in file handling in c

Formatted Input and Output in File Handling (C) In C, file handling functions can also support formatted input and output, just like printf() and scanf() work for standard input and output. For files, we use fprintf() and fscanf() for formatted writing and reading, respectively.

- `fprintf()` is used to write formatted data to the file.

- `fscanf()` is used to read formatted data from the file.
- fprintf(file, “Age: %d\n”, age);
- fscanf(file, “Age: %d\n”, &age);

```
#include <stdio.h>

int main() {
FILE *file;
int age = 30;
float salary = 55000.50;

// Open file for writing
file = fopen(“formatted_data.txt”, “w”);
if (file == NULL) {
printf(“Error opening file for writing.\n”);
return 1;
}

// Write formatted data to the file
fprintf(file, “Age: %d\n”, age);
fprintf(file, “Salary: %.2f\n”, salary);

// Close the file after writing
fclose(file);

// Open file for reading
file = fopen(“formatted_data.txt”, “r”);
if (file == NULL) {
printf(“Error opening file for reading.\n”);
return 1;
}

// Read formatted data from the file
int read_age;
float read_salary;
fscanf(file, “Age: %d\n”, &read_age);
fscanf(file, “Salary: %f\n”, &read_salary);

// Print the read data
printf(“Data read from file:\n”);
printf(“Age: %d\n”, read_age);
printf(“Salary: %.2f\n”, read_salary);
```

```
// Close the file after reading
fclose(file);

return 0;
}
```

Error handling

File Handling Errors: When working with files, always check if a file is successfully opened. If fopen() returns NULL, the file couldn't be opened.

Example:

```
FILE *file = fopen("example.txt", "r");
if (file == NULL) {
printf("Unable to open a file.\n");
return 1; // Exit program with an error code
}
```

File Operation

Basic File Operations in C
In C, file operations are done using the stdio.h library.

1. Opening a File (fopen): To open a file, use the fopen() function. You need to specify the file name and mode (r, w, a, etc.).
Example:

```
FILE *fopen("xyz.txt", "r"); //where xyz.txt is file and r is reading mode of the file
```

modes:
r: Read (file must exist)
w: Write (create file or overwrite)
a: Append (add data at the end)

2. Writing to a File (fprintf, fputs): To write data to a file, use fprintf() or fputs().

3. Reading from a File (fscanf, fgets): To read from a file, use fscanf() or fgets().

4. Closing a File (fclose)

```c
#include <stdio.h>
int main() {
    FILE *file=NULL;
    char ch;
    // Open the file in read mode
    file = fopen("grade.txt", "r");
    // Check if file exists
    if (file == NULL) {
        printf("Error: Unable to open file!\n");
        return 1;
    }
    printf("File content:\n");
    // Read and display characters until EOF (End of File)
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);
    }
    // Close the file
    fclose(file);
    return 0;
```


Rajesh Parajuli

+977-9847546279
parajulirajesh2072@gmail.com

Useful Links

- Home
- Services
- Contact Me
- Terms & Conditions

Worked with

- BidhyaTech
- JK Arts
- MastaSoftsolution
- Ghodaghodi Multiple Campus

Location

Ghodaghodi Municipality-1, Kailali Nepal