1. What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?

→

1. Approach to Problem-Solving

- **Procedural Programming**: Focuses on **functions** (procedures) and the step-by-step execution of instructions. The program is structured around procedures or routines that operate on data.
    - **Approach**: "How to do it" (focus on procedures and step-by-step execution).
- **OOP**: Focuses on **objects** that represent real-world entities, and the interactions between them. The data and the methods (functions) that operate on the data are bundled into objects.
    - **Approach**: "What is it" (focus on objects and their interactions).

2. Data Organization

- **Procedural Programming**: Data is **separate** from functions. Data is typically stored in global variables or passed explicitly between functions.
    - **Example**: A program might have global variables like int x, y; and functions like void calculate() that operate on these global variables.
- **OOP**: Data is encapsulated inside objects. The data and the functions that manipulate the data are grouped together within classes. This makes data more secure and manageable.
    - **Example**: A class Person might have data members name and age, and member functions like getName() and setAge() to manipulate that data.

3. Modularity

- **Procedural Programming**: Code is divided into **functions**. While functions help in organizing the code, there is no concept of encapsulating data within those functions.
    - **Example**: A program is divided into different functions (like input(), process(), and output()).
- **OOP**: Code is divided into **objects** (instances of classes), and each object is a self-contained module with its own data and behavior. This promotes better **modularity** and **reusability**.
    - **Example**: A program might have classes like Car, Bike, and Truck, each of which defines data and methods related to that specific entity.

4. Reusability

- **Procedural Programming**: Reusability is achieved through **functions**. You can call the same function multiple times, but the data is typically shared globally or passed between functions.
    - **Example**: A sorting function that can be reused with different datasets.
- **OOP**: Reusability is achieved through **inheritance** and **polymorphism**. Inheritance allows new classes to inherit properties and methods from existing classes, while polymorphism allows objects of different types to be treated as objects of a common superclass.
    - **Example**: A class Car can be extended to create a ElectricCar class that inherits the behavior of Car but adds specific functionality for electric cars.

5. Data Security

- **Procedural Programming**: Data is typically **not hidden**. It can be accessed and modified directly by any function in the program,

which can lead to potential issues like unwanted side effects and bugs.

- **OOP**: Data is **encapsulated** within objects, and access to the data is controlled via **getters** and **setters**. This ensures better data security and integrity, as external functions cannot directly modify the internal state of an object.
    - **Example**: An object's internal state can be modified only through controlled methods, such as setBalance() and getBalance() in a BankAccount class.

6. State and Behavior

- **Procedural Programming**: The state of the program is typically held in **variables**, and behavior is defined by **functions**. These variables are usually global or passed explicitly between functions.
    - **Example**: A variable total in a function keeps track of the total value.
- **OOP**: The state (data) and behavior (methods) are combined in **objects**. The behavior is represented by methods within the class, and the state is encapsulated as class variables (attributes).
    - **Example**: A BankAccount object might have the state balance and the behavior deposit() and withdraw() methods.

7. Code Maintenance and Flexibility

- **Procedural Programming**: As programs grow in size and complexity, maintaining and modifying the code can become difficult, especially when data and functions are intertwined. Changes to global data often require updates to multiple functions.
- **OOP**: Code is more **modular**, with clear boundaries between objects and classes. This makes it easier to maintain and extend the program. When one part of the program changes (e.g., a class), other parts of the program are less likely to be affected.

8. Program Flow

- **Procedural Programming**: The flow of execution is sequential, with one function calling another. The program typically follows a top-to-bottom approach, and control structures like loops and conditionals govern the program flow.
    - **Example**: A procedural program starts with a main function and calls other functions sequentially.
- **OOP**: The flow of execution is driven by the interaction between **objects**. Each object is responsible for its own state and behavior, and objects interact with each other by calling methods on other objects.
    - **Example**: A Customer object interacts with a BankAccount object to perform actions like deposit() and withdraw().

9. Examples of Programming Languages

- **Procedural Programming**: Languages like **C**, **Pascal**, **Fortran**, and **BASIC** are typically procedural, where the focus is on functions and procedures.
- **OOP**: Languages like **C++**, **Java**, **Python**, **C#**, and **Ruby** are designed with OOP principles, where objects, classes, inheritance, and polymorphism are central concepts.

2.List and explain the main advantages of OOP over POP.

→1. Modularity (Improved Code Organization)

- **OOP**: OOP promotes the concept of **objects** (instances of classes), each representing a module that encapsulates both **data** and **functions**. Each object is a self-contained unit with specific responsibilities. This modular approach makes the code easier to understand, maintain, and reuse.

- **POP**: In POP, the code is generally organized around **functions** and **procedures**, and the data is often handled separately. This makes large programs harder to manage and maintain, as all data and functions are typically global or interdependent, leading to less clear organization.

Advantage: OOP allows developers to work on different parts of a program independently, making it easier to develop and maintain larger applications.

2. Code Reusability (Through Inheritance)

- **OOP**: Inheritance allows new classes (derived classes) to inherit properties and behaviors from existing classes (base classes). This promotes code **reuse** by allowing you to extend or modify the behavior of existing code without changing the original class.
- **POP**: In procedural programming, code reuse is generally achieved through functions, but it lacks the structured mechanism of inheritance. Code is often copied or duplicated across different parts of a program, which can lead to code bloat and difficulties in maintaining consistency.

Advantage: OOP enables the reuse of existing code, which leads to reduced redundancy and better management of code changes.

3. Data Encapsulation (Data Protection)

- **OOP**: **Encapsulation** in OOP allows the internal state of an object to be hidden from the outside world, exposing only a controlled interface (through getter and setter methods). This ensures that the object's data cannot be directly modified, protecting it from unwanted changes and making it more secure and reliable.
- **POP**: In procedural programming, the data is typically global or passed between functions, which can result in **uncontrolled**

access and modification of data. This lack of control increases the chances of introducing errors or bugs.

Advantage: OOP ensures better protection and integrity of data, as access is strictly controlled.

4. Flexibility and Extensibility (Polymorphism and Inheritance)

- **OOP**: **Polymorphism** allows objects of different classes to be treated as objects of a common superclass. This allows you to write more **generic** and **flexible** code. Additionally, OOP supports easy **extension** of code (through inheritance) without modifying existing classes.
- **POP**: In procedural programming, it is harder to extend or modify code because the procedures are usually closely coupled with data. Changes to the program often require significant modifications to existing code.

Advantage: OOP makes it easier to extend and modify the system as requirements evolve, leading to more flexible and scalable applications.

5. Improved Maintainability and Manageability

- **OOP**: Since OOP promotes modularity, encapsulation, and abstraction, it makes the software easier to **maintain** and **manage**. Each object can be modified or debugged independently without affecting the rest of the program. This leads to better **separation of concerns**.
- **POP**: In procedural programming, a change in one part of the program can often require modifications across multiple functions, leading to difficulties in maintenance, especially in large systems.

Advantage: OOP's modular structure and data encapsulation simplify maintenance, debugging, and refactoring.

6. Abstraction (Hiding Complexities)

- **OOP**: **Abstraction** in OOP allows complex systems to be represented through simpler, higher-level interfaces. It hides the internal details and provides a simplified interaction with objects. For example, you don't need to know the details of how a car's engine works to drive the car.
- **POP**: Procedural programming doesn't inherently support abstraction as effectively. While functions can abstract operations, the focus is typically on procedural flow, and it's harder to hide the complexity of large systems.

Advantage: OOP allows developers to work at a higher level of abstraction, focusing on what needs to be done rather than how it is done.

3.Explain the steps involved in setting up a C++ development environment.

→

- Install a C++ compiler (GCC, MinGW, MSVC, Clang).
- Install an IDE or text editor (Visual Studio, Code::Blocks, CLion, VSCode).
- Create a new project or workspace for your C++ code.
- Set up environment variables (if needed).
- Write your first C++ program.
- Compile and run the program using your IDE or command line.
- Debug your program using the built-in debugger (if needed).
- Install additional libraries or frameworks as required.

- Regularly update your development tools to ensure optimal performance.

4.What are the main input/output operations in C++? Provide examples.

→

Main Input/Output Operations in C++:

1. **Standard Output (cout)**: Used for output to the console.
2. **Standard Input (cin)**: Used for input from the console.

Key Operators for I/O in C++:

- **<<**: Stream insertion operator, used for output.
- **>>**: Stream extraction operator, used for input.

  Exmple :

  Include<iostream>

  Using namespace std;

  Main()

  {

  Int a,b;

  Cout<<”Enter any two number :;

  Cin>>a>>b;

  Cout<<”a =”<<a<<”b=”<<b;

  }

5.What are the different data types available in C++? Explain with examples.

→

- Data type: data type is type of data and variable store.

  Datatype is two type :

  1.primitive datatype.

  2.non-primitive datatype.

  Primitive datatype : primitive datatype is provide language.

  Ex : int , float , char, etc…

  Non-primitive datatype : non-primitive datatype is provide developer.

  Ex : string,array , structure etc…

Integer : declare %d store the positive value small range.

Float : declare %f store the decimal value small range.

Long integer : declare %d store the positive value high range.

double : declare %f store the decimal value high range.

Character : declare %c store the single character value.

String : declare %s store the group of character value.

6.Explain the difference between implicit and explicit type conversion in C++.

→

1. Implicit Type Conversion (Automatic Type Conversion)

- **Definition**: Implicit type conversion, also known as **type coercion**, happens automatically when the compiler converts one data type to another without any explicit instruction from the programmer.
- **When It Happens**: It occurs when a value of one data type is assigned to a variable of another type, and the conversion is safe and does not lead to any loss of data or errors.
- **How It Works**: The compiler performs the conversion automatically, and it usually happens when converting from a smaller data type to a larger one (e.g., from int to float, or from char to int).
- **Example**:

  cpp
  Copy code
  int i = 10;
  double d = i; // Implicit conversion from int to double

  In this example, the int variable i is automatically converted to double when assigned to d.

- **Common Conversions**:
  - int to float or double
  - char to int
  - float to double
- **Advantages**:
  - Reduces the need for manual conversion.
  - Simplifies code when safe conversions are required.

2. Explicit Type Conversion (Type Casting)

- **Definition**: Explicit type conversion, also known as **type casting**, requires the programmer to manually convert a variable from one type to another using a cast operator.
- **When It Happens**: It happens when the programmer specifically requests the conversion, typically when converting from a larger data type to a smaller one, or when the conversion may result in a loss of data or precision.
- **How It Works**: The programmer uses casting operators (static_cast, dynamic_cast, const_cast, or reinterpret_cast) to explicitly tell the compiler how to perform the conversion.
- **Example**:

  cpp
  Copy code
  double d = 10.5;
  int i = (int)d; // Explicit conversion from double to int

  Here, the double value is explicitly cast to int, which may result in the loss of the decimal part.

- **Common Conversions**:
  - double to int (explicit casting needed because it may lose precision)
  - void* to specific pointer types
  - int to char (can result in truncation of the value)
- **Advantages**:
  - Provides more control over type conversion, especially when precision or data loss is a concern.
  - Useful for complex conversions that cannot be done implicitly.

7.What are the different types of operatorsin C++? Provide examples of each.

→

arithmetic,

relational,

logical,

assignment,

increment/decrement,

bitwise,

conditional operators.

- Operator :Operator is use in two oprants.operator is symbol perform the to get the mathamatics operation.

  Multiple operator in c language

- arithmetic operator: Addition , subtraction , multiplication , division, modulo etc. perform the mathematical operation use arithmetic operator.

  Ex :

addition	+
subtraction	-
multiplication	*
division	/
modulo	%

- relational operator: perform the two value comparison use relational operator.

  Ex :

Double equal	==
Not equal	!=
Grater than	>
Less than	<
Less than equal	<=
Grater than equal	>=

- logical operator : perform the two or more condition check use logical operator.

  Ex :&&(and) , ||(or) , !(not)

And	&&
Or	||
Not	!

- assignment operator : perform the assign value use assignment operator.

  Ex: += , -+, *= , /=

a-=b	a=a-b
a*=b	a=a*b
a/=b	a=a/b
a+=b	a=a+b

- increment/decrement :increment operator value+1,decrement operator value-1

  Prefix : ++a , --a postfix : a++ , a--

- bitwise operator : ex : & , | , << , >>

8. Explain the purpose and use of constants and literals in C++.

→

1. Constants in C++

Definition: A **constant** is a value that cannot be changed once it is assigned. Constants are useful when you want to define a value that remains the same throughout the execution of the program.

- **Purpose**: Constants help improve code readability, maintainability, and reliability. By using constants, you can prevent accidental changes to values that should remain fixed, and you can give meaningful names to those values.
- **Use**:
    - Constants are usually defined using the const keyword or through the #define preprocessor directive.
    - They are typically used for values that are used repeatedly in the code and should not be altered (e.g., mathematical constants like PI, maximum limits, configuration values).

2. Literals in C++

Definition: A **literal** is a fixed value written directly in the source code. Literals represent basic values like numbers, characters, or boolean values. They are the most direct form of data representation.

- **Purpose**: Literals are used to represent constant values directly in the code. They provide a simple and clear way to specify fixed values.
- **Use**:
    - Literals are used to assign values to variables, parameters, or expressions directly in the code. For example, a numeric literal (5) or a string literal ("Hello, World!") can be used to initialize a variable or in an expression.

9.What are conditional statements in C++? Explain the if-else and switch statements.

→

Conditional Statements in C++

Conditional statements in C++ are used to execute specific blocks of code based on whether a given condition (or set of conditions) is true or false. These statements allow the program to make decisions and control the flow of execution based on certain criteria.

- switch statement: To make menu driven code.Never use relational op. by this. (n==1).switch can be used with only int, char, double data types.

  switch can be used with the keywords :

  switch, case, break, default.

  Syntex : switch(choice)

  {

  Case 1: // statement

  Break;

  Case 2 : //statement

  Break;

  Default : //statement

- if_else statement : if else is use to check the condition is true or false .

  Syntex : if(condition)

  {

  //statement

```
}
else
{
        //statement
}
```

Ex:

```
#include<stdio.h>
Main(){
Int a=18;
If (a>18)
{
Printf(“eligible for voting .”);
}
else
{
Printf(“you are not eligible for voting .”);
}
```

10.What is the difference between for, while, and do-while loops in C++?

→

- **While loop :** while(condition) { execute the code } .while loop condition is true loop is execute and condition is false loop is not execute. while loop is entry control loop, entry control loop is check condition before execute code. Exit the loop condition is false.

e.g :

While loop :

```
I = 1;                  //(initialization)

While(I <= 5)           //condition
{
printf ("%d", i);   //execute code
I++;                    // increment/decrement
}
```

- **For loop :** for(initialization , condition , increment/decrement). for loop condition is true loop is execute and condition is false loop is not execute. for loop is entry control loop, entry control loop is check condition before execute code. Exit the loop condition is false.

e.g :

For loop :

```
For
(I=1(initialization; I<=5(condition);i++(increment/decrement))
{
```

```
printf (“%d” ,i);            // execute code

 }
```

- **Do while loop :** do { execute the code } while(condition) . do while loop condition first time execute and after condition is true loop is execute and condition is false loop is not execute. do while loop is exit control loop, exit control loop is execute code before check condition. Exit the loop condition is false.

e.g :

Do while loop :

```
I = 1;(initialization)
 Do   {
      printf (“%d” ,i);            // execute code
      I++;                                  // increment/decrement
      } While (I <= 5);           // (condition)
```

11. **How are break and continue statements used in loops? Provide examples.**

→

Break statement : Break is keyword and break keyword use the break the code, rest of the code will not executed .

Example :

```
#include<stdio.h>
Main()
{
 Int I;
```

```
for(i=0;i<10;i++)
{
        If(i==6)
        Break;
        Printf(“%d”,I);
}
```

Continue statements : The continue statement is used within loops to skip the remaining statements in the current iteration and proceed to the next iteration of the loop.

Example :

```
#include <stdio.h>

main()

{

        for (int i = 1; i<= 10; i++)

        {

                if (i % 2 == 0)

                        {

                         continue; // Skip even numbers

                        }

                printf("%d ", i);

         }

}
```

12.Explain nested control structures with an example.

→

1. nested if statement : check multiple condition in one time than used to nested if condition within if condition (if into if)

   Syntex : if(condition)

```
{
If(condition)
{
    //statement
}
 else
{
    //statement
}
else
{
    //statement
}
```

Ex :

```
#include <iostream>
 using namespace std;
```

```
main()
{
int age = 20;
 bool hasPermission = true;
if (age >= 18)
 {
        if (hasPermission)
         {
                cout << "You are allowed to enter." <<
                endl;
         }
        else
        {
                cout << "You are not allowed to enter due
                to lack of permission." << endl;
         }
    }
 else
{
cout << "You are not old enough to enter." << endl;
```

} }

13. What is a function in C++? Explain the concept of function declaration, definition, and calling.

→

Function is block of code use the function () round breaket , function is two type :

1) Build in function : build in function is already has a c language provide .it is not created by developer.
2) User define function :user define function is provide by developer.

Mainly four type for user define :

1. Fun with no argument no return value.

2. Fun with argument but no return value.

3. Fun without argument with return value.

4. Fun with arg. with return value.

- Function mainly three type :
  - Function declaration
  - Function calling
  - Function definition

Example :

```
#include<stdio.h>


void hello();  //1. fun. declaration
```

```
main()
{
        hello(); //2. fun. calling


}


void hello()  //3. fun. definition
{
        printf("\n\n\t Hello() is called.");
}
```

14. What is the scope of variables in C++? Differentiate between local and global scope.

→

Scope of Variables in C++

In C++, the scope of a variable refers to the region of the program where the variable is accessible or valid. The scope determines the lifetime and visibility of the variable, i.e., where it can be used or modified.

C++ has two primary types of variable scopes:

1. Local Scope
2. Global Scope

1. Local Scope

A **local variable** is a variable that is declared within a function or block of code (enclosed by curly braces {}). Its scope is limited to the function or block where it is declared. Once the execution leaves that scope, the variable is no longer accessible, and its memory is freed.

- **Visibility**: A local variable is only visible within the function or block where it is declared.
- **Lifetime**: The lifetime of a local variable is tied to the execution of the function or block in which it is declared. The variable is created when the function or block is entered, and it is destroyed when the function or block exits.
- **Default Value**: Local variables are not automatically initialized, and they hold garbage values until explicitly initialized.

2. Global Scope

A **global variable** is declared outside all functions, usually at the top of the program. Its scope extends across the entire program, meaning it is accessible by all functions after it is declared.

- **Visibility**: A global variable is visible throughout the entire program, including all functions that appear after its declaration.
- **Lifetime**: The lifetime of a global variable is the entire runtime of the program. It is created when the program starts and is destroyed when the program ends.
- **Default Value**: Global variables are automatically initialized to 0 for numeric types, false for boolean types, and nullptr for pointers if not explicitly initialized.

15. Explain recursion in C++ with an example.

→Recursion in C++

Recursion in C++ refers to the process in which a function calls itself in order to solve a problem. A recursive function breaks down a problem

into smaller, more manageable sub problems. Each recursive call handles a part of the problem, and eventually, the base case is reached, which stops further recursion.

Example :

```
#include <iostream>

using namespace std;

int factorial(int n) {

   if (n == 0) {

      return 1;

   }

   else {

      return n * factorial(n - 1);

   }

}

 main() {

   int number;

   cout << "Enter a number to find its factorial: ";

   cin >> number;

   cout << "Factorial of " << number << " is " << factorial(number) <<
endl;
```

```
  return 0;
}
```

16.What are function prototypes in C++? Why are they used?

→Function Prototypes in C++ :

A **function prototype** in C++ is a declaration of a function that provides information about the function's name, return type, and parameters (if any) without providing the full function body. It is typically placed before the main() function or at the beginning of the program.

A **function prototype** gives the compiler enough information about how to call the function even before the actual definition of the function is encountered in the code.

Syntax of a Function Prototype :

return_type function_name(parameter_list);

- **return_type**: The type of value the function will return (e.g., int, double, void, etc.).
- **function_name**: The name of the function.
- **parameter_list**: A list of parameters (if any) that the function takes, enclosed in parentheses. If there are no parameters, you can either leave the parentheses empty or explicitly use void.

Function Prototype :

-Enables Function Calls Before Function Definitions

-Type Checking for Arguments

-Better Code Organization

-Allowing Forward Declaration

17. What are arrays in C++? Explain the difference between single-dimensional and multidimensional arrays.

→ Array : Array is a collection of elements/values with similar data type.

- Types of Arrays :

  1) One Dimentional Array

  e.g int arr[5];

  2) Two Dimentional Array

  e.g int arr[3][3];

  3) Multi Dimentional Array

  e. g int arr[4][3][3];

- Differentiate between one-dimensional and multi-dimensional arrays with example.
  Diffrence :
  →one dimensional array stored single line multiple element, output execute series type.

  →A multi-dimensional array is essentially an array of arrays. The most common type is the two-dimensional array, often used to represent matrices.

- Multi- dimensional arrays

```
#include<iostream>
main()
{
        int mat[2][3][3];
        int m, r, c;

        for(m=0;m<2;m++)
        {
                Cout<<"\n\n\t Matrix : "<< m;
                for(r=0;r<3;r++)
                {
                        for(c=0;c<3;c++)
                        {
                                Cout<<"\n\n\t Input mat : "<<m<<
                                r<<c;
                                cin>>mat[m][r][c];
                        }
                }
        }

        for(m=0;m<2;m++)
        {
                Cout<<"\n\n\t Matrix  : \n\n"<<m;
                for(r=0;r<3;r++)
                {
                        for(c=0;c<3;c++)
                        {
```

```
                Cout<<mat[m][r][c];
            }
            Cout<<"\n";
        }
    }}
```

18.Explain string handling in C++ with examples.

→ String handling functions are a set of standard library functions in C, defined in <string.h>. These functions provide essential tools for working with strings (character arrays). Here's a detailed explanation of common string-handling functions:

1. strlen()

- **Purpose:** Computes the length of a string (excluding the null terminator \0).
- **Prototype:**size_tstrlen(const char *str);
- **Use case:** Determine the size of a string for processing or allocating memory.

  Example :

```
#include <iostream>

#include <string.h>


main() {

  char str[] = "Hello, World!";
```

```
cout<<"Length of the string: \n" <<strlen(str);

}
```

2. strcpy()

- **Purpose**: Copies the source string (including the null terminator) to the destination string.
- **Prototype**: char *strcpy(char *dest, const char *src);
- **Use case**: Duplicate or overwrite a string.

  Example :

```
#include <iostream>

#include <string.h>


main() {

   char src[] = "Welcome!";

   char dest[20];


strcpy(dest, src);

cout<<"Destination String: \n"<<dest;


}
```

3. strcat()

- **Purpose**: Concatenates (appends) the source string to the end of the destination string.
- **Prototype**: char *strcat(char *dest, const char *src);
- **Use case**: Combine two strings.

  Example :

  ```
  #include <iostream>

  #include <string.h>


  main() {

     char str1[50] = "Hello";

     char str2[] = ", World!";


  strcat(str1, str2);

  cout<<"Concatenated String:\n"<<str1;

  }
  ```

  4. strcmp()

- **Purpose**: Compares two strings lexicographically.
- **Prototype**: int strcmp(const char *str1, const char *str2);
- **Return Values**:
  - **< 0**: str1 is less than str2.
  - **0**: str1 is equal to str2.

- **> 0**: str1 is greater than str2.

Use case: Compare strings for sorting or equality checks.

Exmple :

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

main() {
   char str1[] = "apple";
   char str2[] = "banana";

   int result = strcmp(str1, str2);
   if (result < 0)
cout<<" comes before \n"<<str1<<str2;
   else if (result > 0)
cout<<" comes after \n"<<str1<<str2;
   else
cout<<" is equal to \n"<< str1<< str2;
   }
```

5. strchr()

- **Purpose**: Finds the first occurrence of a character in a string.

- **Prototype**: char *strchr(const char *str, int c);
- **Return Value**: A pointer to the first occurrence of the character, or NULL if not found.
- **Use case**: Locate specific characters in a string.

  Example :

```
#include <iostream>

#include <string.h>


main() {

   char str[] = "Hello, World!";

   char *ptr = strchr(str, 'W');


 if (ptr)

cout<<"Character 'W' found at position: \n"<< ptr - str;

else

cout<<"Character noundot f.\n";

   }
```

19. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

→arrays can be initialized in several ways. Arrays are a collection of elements of the same type, and they can be either

1-dimensional (1D) initialized :

```
#include <iostream>
using namespace std;
int main()
{
// Static initialization with known
int arr[5] = {1, 2, 3, 4, 5};
 // Printing the array elements
for(int i = 0; i < 5; i++)
 {
        cout << arr[i] << " ";
 }
cout << endl;
return 0;
}
```

2-dimensional (2D) initialized :

```
#include <iostream>
 using namespace std;
```

```
 int main()
{
 // Static initialization of a 2D array
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}
};
 // Printing the 2D array elements
for(int i = 0; i < 2; i++)
 {
        for(int j = 0; j < 3; j++)
        {
                cout << arr[i][j] << " ";
         } cout << endl;
 } return 0;
 }
```

20.Explain string operations and functionsin C++.

→

- **gets()** : to read the string with space.
- **puts()** : to print the string
- **strlen()** : to find out the length of the string

- **strrev()** : o reverse the string
- **strlwr()** : to convert into lower case
- **strupr()** : to convert into upper case
- **strcpy()** : to copy one string to another.
- **strcat()** : To concate two strings.
- **strcmp()** : To compare two strings. (by default refer the case)
- **stricmp()** : To compare two strings. (by ignoring the case)

21. Explain the key concepts of Object-Oriented Programming (OOP).

→

1. Class and Object

- **Class** : A class is a blueprint or prototype for creating objects. It defines a set of properties (data members) and behaviors (member functions or methods) that the objects of the class will have.

  Exmaple : class class_name

- **Object** : An object is an instance of a class. It represents a specific entity created based on the class blueprint, having its own set of values for the properties.

  Exmaple : class_name object_name ;

2. Encapsulation

Encapsulation is the concept of **bundling the data (attributes)** and the methods (functions) that operate on the data into a single unit or class.

It is the principle of restricting direct access to some of an object's components and only allowing access through public methods.

- **Access Control**: Encapsulation is achieved using
- **access specifiers**:
    - **Private**: Members are accessible only within the class.
    - **Public**: Members are accessible from outside the class.
    - **Protected**: Members are accessible within the class and its derived classes.

3. **Abstraction**:

- Abstraction is the process of hiding the complex implementation details and exposing only the necessary and relevant parts of the object to the outside world.
- It allows a programmer to focus on interactions at a higher level without needing to understand all the inner workings.

4. **Inheritance**:

- Inheritance allows a class to inherit properties and behaviors from another class, which is called a **parent** or **superclass**.
- The class that inherits is called the **child** or **subclass**.
- Inheritance promotes code reusability and allows for the creation of hierarchical relationships between classes. Subclasses can extend or override methods from the superclass.

5. **Polymorphism**:

- Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying forms (data types).
- There are two types of polymorphism:
    - **Method Overloading**: Having multiple methods with the same name but different parameters within the same class.

- **Method Overriding**: A subclass provides its own implementation of a method that is already defined in the superclass.

22. What are classes and objects in C++? Provide an example

→

- **Class in C++**

A **class** is a user-defined blueprint or prototype for creating objects. It defines the attributes (data members) and behaviors (member functions or methods) that the objects of the class will have.

A class does not allocate memory or create objects directly. Instead, objects are created from the class as instances.

- **Object in C++**

An **object** is an instance of a class. Once a class is defined, you can create objects from it. Objects are the real entities that hold actual data and can invoke the functions defined in the class.

- **Key Components of a Class in C++:**

- **Data members**: Variables that hold the state of the object.
- **Member functions**: Functions that define the behavior of the object.
- **Constructor**: A special function that is called when an object is created, initializing the object's data members.
- **Destructor**: A special function that is called when an object is destroyed.

  Example :

  #include<iostream>

```
using namespace std;


class rectangle
{
        int lenght,width;
         public:


                void get()
                {
                        cout<<"\n\n\tEnter the lenght :";
                        cin>>lenght;
                        cout<<"\n\n\tEnter the width :";
                        cin>>width;


                }
                void print()
                {
                        cout<<"\n\n\tArea of ractangel:"<<lenght*width;
                }
```

```
};
main()
{
        rectangle a;
        a.get();
        a.print();
}
```

23. What is inheritance in C++? Explain with an example.

→Inheritance in C++

Inheritance is one of the core concepts of Object-Oriented Programming (OOP) that allows a class to inherit attributes and methods from another class. In C++, inheritance is used to create a new class (called a **derived class**) based on an existing class (called a **base class**), promoting **code reuse** and **extensibility**.

Key Points about Inheritance in C++:

- The **base class** is the class from which properties and behaviors are inherited.
- The **derived class** is the class that inherits properties and behaviors from the base class.
- The derived class can add new members (data or functions) or override existing ones.
- **Access control** can be used to determine which members of the base class can be accessed in the derived class.

Types of Inheritance:

1. **Single Inheritance**: A derived class inherits from only one base class.
2. **Multiple Inheritance**: A derived class inherits from more than one base class.
3. **Multilevel Inheritance**: A class is derived from another derived class.
4. **Hierarchical Inheritance**: Multiple derived classes inherit from a single base class.
5. **Hybrid Inheritance**: A combination of more than one type of inheritance.

   Example :

```
#include<iostream>

using namespace std;


class parson

{

        protected:

                string s_name , t_name;

        public:

                void get();

                void get1();

};
```

```
void parson::get()
{
	cout<<"enter the s_name:";
	cin>>s_name;
}

void parson::get1()
{
	cout<<"\nenter the t_name:";
	cin>>t_name;
}

class student : public parson
{
	protected:
		int age;
		string course;
	public:
```

```
        void put();
};

void student::put()
{
    cout<<"enter the age:";
    cin>>age;
    cout<<"enter the course:";
    cin>>course;

    cout<<"\ns_name:"<<s_name;
    cout<<"\nage:"<<age;
    cout<<"\ncourse:"<<course;
}

class teacher : public parson
{
    protected:
        int id;
```

```
        int salary;
    public:
        void put2();
        void print();
};

void teacher::put2()
{
    cout<<"enter the id:";
    cin>>id;
    cout<<"enter the salary:";
    cin>>salary;
}

void teacher::print()
{

    cout<<"\nt_name:"<<t_name;
    cout<<"\nid:"<<id;
```

```
        cout<<"\nsalary:"<<salary;


}


main()
{
        student obj;
        obj.get();
        obj.put();
        teacher ob;
        ob.get1();
        ob.put2();
        ob.print();
}
```

24.What is encapsulation in C++? How is it achieved in classes?

→

- **Encapsulation in C++**

Encapsulation is one of the four fundamental Object-Oriented Programming (OOP) principles. It refers to the concept of bundling the data (attributes) and methods (functions) that operate on the data into a single unit known as a class. Additionally, encapsulation restricts

direct access to some of an object's components, which helps prevent unintended interference and misuse of the object's data.

The main goal of encapsulation is to:

1. **Protect the data**: By hiding the internal state of an object and allowing controlled access via public methods.
2. **Control access**: Providing controlled access to data through methods (getters and setters) rather than directly exposing the data.

- **Achieving Encapsulation in C++ with Classes**

In C++, encapsulation is primarily achieved by using **access specifiers** to define the visibility of class members (attributes and methods). These access specifiers are:

- **private**: Members declared as private cannot be accessed directly from outside the class. This ensures that sensitive data is protected and cannot be modified directly from outside the class.
- **public**: Members declared as public are accessible from outside the class. These are the methods that allow interaction with the object.
- **protected**: Members declared as protected are similar to private members but can be accessed in derived classes (subclasses).