1). Explain why OOP is better than POP. Point out the issues with POP.

-> OOP is better than POP because OOP supports many features that are not supported in POP. OOP has additional functionalities and features that can relate and solve real-time world problems and it has the features to secure and manipulate the data in a more effective and efficient manners compared to POP. The features that make OOP better than POP are as:

- **Modularity**: OOP allows for modular code through the use of classes, making it easier to manage and maintain. Each class can be developed, tested, and debugged independently.

- **Reusability**: With OOP, code reusability is higher. Once a class is written, it can be reused across multiple programs without modification.

- **Encapsulation**: OOP promotes encapsulation, which means keeping data and methods that manipulate that data together in one place. This helps in protecting data from unintended modification.

- **Abstraction**: OOP supports data abstraction by allowing complex systems to be broken down into simpler, more manageable objects.

- **Inheritance**: OOP enables inheritance, which allows new classes to inherit properties and methods from existing classes, promoting code reuse and reducing redundancy.

The issues with POP are as:

- It doesn't have the provision of data access control and doesn't have a secure way of hiding the data.
- It doesn't have the features like polymorphism, virtual function and encapsulation.
- Inheritance and Virtual functions are not supported in POP.
- POP follows the traditional way of writing codes into manageable parts known as modules which doesn't ensure the integrity and confidentiality of data.
- It is difficult to develop complex and real-time world applications in POP.
- POP has issues with code reusability and data abstraction.

2). What are the advantages and disadvantages of OOP? Explain in brief the history of C++.

Object-Oriented Programming (OOP) offers several advantages that make it a popular choice for software development.Here are some of the advantages of OOP:-

- **Modularity:** OOP breaks down complex systems into smaller, self-contained units called objects. This makes the codebase easier to understand, maintain, and modify.
- **Reusability:** OOP promotes code reuse through inheritance. Inheritance allows you to create new classes (subclasses) that inherit properties and behaviors from existing classes (superclasses).

- **Flexibility and Scalability:** OOP allows you to easily add new features or modify exis
- **Improved Code Organization:** OOP promotes a structured approach to programming, leading to well-organized and readable code. This improves collaboration among developers and simplifies maintenance.
- **Other Advantages:** OOP also offers benefits like improved security through data hiding, easier debugging due to modularity, and the ability to model real-world entities using objects.

The disadvantages of OOP are:

- **Larger Program Size**: OOP programs tend to have larger codebases due to the creation of many small classes and objects, which can lead to increased memory usage.

- **Slower Execution**: Due to the extra layers of abstraction and dynamic method resolution, OOP programs can run slower compared to their procedural counterparts.

- **Complexity**: OOP can introduce additional complexity to the design and implementation of software, which can be overwhelming for beginners.

- **Performance Overhead**: The abstraction layers and the dynamic features of OOP (like polymorphism) can sometimes lead to performance overhead compared to procedural programming.

History of C++

C++'s story starts in the late 1970s with a Danish computer scientist named Bjarne Stroustrup at Bell Labs. He was working on his Ph.D. thesis and wanted a language that offered:

- The efficiency and familiarity of C for system programming
- High-level features like object-oriented programming for better code organization

Stroustrup based C++ on C, adding object-oriented concepts like classes and inheritance. Initially called "C with Classes," it was later renamed C++. The first public release came around 1983.C++ gained popularity due to its ability to handle both high-level and low-level programming, making it suitable for various tasks. Here are some milestones:

- Standardization: In 1989, efforts began to standardize C++, resulting in the first official standard in 1998. This ensured consistency across different compilers.
- Growth and Evolution: Since then, C++ has continued to evolve with new features added through regular standardization updates. The latest standard is C++20, with C++23 planned for release soon.

Today, C++ remains a widely used language for various applications, including:

- System programming (operating systems, device drivers)
- Game development
- High-performance computing

- Embedded systems

Its blend of efficiency and rich features makes C++ a powerful tool for programmers across different domains.

3). Differentiate between OOP and POP

The differences between OOP and POP are tabulated as:

POP	OOP
1) Emphasis is given to procedure.	1) Emphasis is given as data.
2) Programs are divided into function.	2) Programs are divided into Objects.
3) Generally data cannot be hidden,	3) Data can be hidden so that non-member cannot access them
4) Data move from function to Function.	4) Data and function are tied together.Only related function can access them.
5) Follow top-down approach of program design.	5) Follow bottom-up approach of program design.

C and C++ (In points)

C	C++
C language was developed by Dennis Ritchie.	C++ language was developed by Bjarne Stroustrup.
C is a structured programming language.	C++ supports both structural and object-oriented programming language.
C is a subset of C++.	C++ is a superset of C.
In C language, data and functions are the free entities.	In the C++ language, both data and functions are encapsulated together in the form of a project.
C does not support the data hiding. Therefore, the data can be used by the outside world.	C++ supports data hiding. Therefore, the data cannot be accessed by the outside world.
C supports neither function nor operator overloading.	C++ supports both function and operator overloading.
In C, the function cannot be implemented inside the structures.	In the C++, the function can be implemented inside the structures.
Reference variables are not supported in C language.	C++ supports the reference variables.
C language does not support the virtual and friend functions.	C++ supports both virtual and friend functions.

Differences between C and C++ Programs

Here, programs in both C and C++ displays the message “hello world”

C Program:

```c
#include <stdio.h>
int main()
{
  printf("hello world");
  return 0;
}
```

C++ Program

```cpp
#include <iostream>
using namespace std;
int main()
{
  cout<<"hello world";
  return 0;
}
```

4).Explain the features of OOP. Describe in brief the lexical components of C++.

Object-Oriented Programming (OOP) is a programming paradigm that revolves around objects. These objects are self-contained entities that group data (attributes) with the code that manipulates that data (methods). The key features of OOP are explained as:

- **Encapsulation**:

  ➔ Encapsulation is the practice of keeping an object's data (attributes) and methods (functions) that manipulate the data together in a single unit or class. It protects the internal state of the object and prevents unauthorized access, enhancing security and robustness.

- **Abstraction**:

  ➔ Abstraction involves hiding the complex implementation details of a system and exposing only the essential features. It simplifies interaction with complex systems by providing a clear and simplified interface.

- **Inheritance**:

  ➔ Inheritance allows a new class (derived class) to inherit properties and methods from an existing class (base class). It promotes code reuse and establishes a natural hierarchical relationship between classes.

- **Polymorphism**:

  ➔ Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. The two main types are compile-time (method overloading) and runtime (method overriding) polymorphism. It enhances flexibility and maintainability by allowing the same interface to be used for different underlying forms (data types).

- **Class**:

    ➔ A class is a blueprint or template for creating objects. It defines the attributes and methods that the created objects will have. It helps in organizing and structuring code in a modular and reusable manner.

These core features along with others like constructors, destructors, and method overloading work together to create well-structured, maintainable, and reusable code in OOP.

The lexical components of C++ are the basic building blocks that make up your C++ program before it's even processed by the compiler. These components are like the individual words and punctuation marks that come together to form sentences in a language. Here's a breakdown of the main lexical components:

1. **Keywords:** These are reserved words with predefined meanings in C++. Examples include int, for, if, class,public, and many more. You cannot use keywords for any other purpose in your code.
2. **Identifiers:** These are user-defined names given to variables, functions, classes, and other entities in your program.Identifiers must follow specific naming rules (e.g., start with a letter or underscore, contain letters, numbers, and underscores).
3. **Literals:** These represent fixed values directly included in your code. Examples include integer literals (e.g., 10,-25), floating-point literals (e.g., 3.14, 1.2e-5), character literals (e.g., 'a', '\n'), string literals (e.g., "Hello, world!").
4. **Operators:** Operators are symbols that perform operations on values. Examples include arithmetic operators (+, -,*, /), comparison operators (==, !=, <, >), logical operators (&&, ||, !), and many more.
5. **Punctuators:** Punctuators are special symbols used to separate statements, control flow, and define code structure.Examples include commas (,), semicolons (;), parentheses (), {}, []), braces, and brackets.
6. **Comments:** Comments are lines of text ignored by the compiler. They are used to provide explanations and notes within the code to improve readability and maintainability. There are single-line comments (//) and multi-line comments (/* ... */).
7. **Whitespace:** Spaces, tabs, and newlines are generally ignored by the compiler but improve code readability.However, whitespace can be significant in certain contexts, like separating tokens.

These lexical components are combined by the compiler to form meaningful statements, expressions, and ultimately, your entire C++ program. Understanding these building blocks is essential for writing correct and well-structured C++ code.

5). What is type conversion in C++? Explain explicit type conversion with the program.

-> Type conversion in C++ refers to the process of transforming a value from one data type to another. This can be done implicitly (automatically by the compiler) or explicitly (by the programmer using a cast operator).

There are two main types of type conversion:

1. **Implicit Conversion (Automatic Type Promotion):**

- The compiler performs this conversion automatically when an expression involves operands of different data types.
- Generally, the compiler promotes the operand with the smaller range to the data type of the operand with the larger range to avoid data loss.
- For example, adding an int (like 10) and a float (like 3.14) will implicitly convert the int to a float before performing the addition.

2. **Explicit Conversion (Casting):**
   - The programmer forces a conversion from one data type to another using a cast operator.
   - This is useful when you want to control the conversion process or convert between data types that might not be implicitly promoted (e.g., converting a double to an int might truncate the decimal part).
   - C++ offers different cast operators for various scenarios, such as static_cast, dynamic_cast, const_cast,and reinterpret_cast.

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

int main() {
  // 1. Converting int to double (no data loss)
  int age = 25;
  double salary = static_cast<double>(age); // Cast age (int) to salary (double)
  cout << "Salary (double): " << salary << endl;

  // 2. Converting double to int (potential data loss)
  double pi = 3.14159;
  int rounded_pi = static_cast<int>(pi); // Cast pi (double) to rounded_pi (int) - truncates de
  cout << "Rounded Pi (int): " << rounded_pi << endl;

  // 3. Converting char to int (represents ASCII value)
  char initial = 'A';
  int ascii_value = static_cast<int>(initial); // Cast initial (char) to ascii_value (int) - ho
  cout << "ASCII value of 'A': " << ascii_value << endl;

  return 0;
}
```

6). What is a namespace? Why do we need this? Write a program to demonstrate namespace using ‘using’ keyword and scope resolution operator.

-> A **namespace** is a feature in C++ that provides a way to group logically related identifiers such as variables, functions, classes, and objects under a single name. It is used to organize code into a coherent structure and to avoid name conflicts in larger programs and libraries.. The reasons we need namespace are for:

- **Avoid Naming Conflicts**:
    - In large projects or when using multiple libraries, different components might use the same names for variables, functions, or classes. Namespaces help avoid conflicts by distinguishing these identifiers.
- **Code Organization**:
    - Namespaces provide a way to group related code together logically, making the codebase easier to understand and maintain.
- **Modularity**:
    - Namespaces allow for the modular design of code. Different modules or libraries can be developed independently without worrying about naming collisions.

Program to demonstrate namespace using 'using' keyword and scope resolution operator.

```
#include <iostream>

using namespace std;

char c = 'a';     // global variable

int main() {
  char c = 'b';   //local variable

  cout << "Local  variable: " << c << "\n";
  cout << "Global variable: " << ::c << "\n";  //using scope resolution operator

  return 0;
}
```

7). How do we define user defined constants? Show by program.

There are three main ways to define user-defined constants in C++:

1. **Using the const keyword:** This is the most common and recommended approach for constants in C++.

   Syntax: const data_type constant_name = value;

2. **Using #define preprocessor directive (C-style):** This approach is inherited from C and is generally less preferred in C++ due to potential drawbacks.

Syntax: #define CONSTANT_NAME value

3. **Using constexpr keyword (C++11 and later):** This approach provides compile-time constant evaluation, offering some benefits over const.

   Syntax: constexpr data_type constant_name = expression;

Program to define user defined constants

Using the const keyword:

```
#include <iostream>
const int MAX_STUDENTS = 30;  // Maximum number of students
const double GRAVITY = 9.81;   // Gravitational constant (m/s^2)
int main() {
  int numStudents = 25;
  if (numStudents <= MAX_STUDENTS) {
    std::cout << "Class size is within limit." << std::endl;
  } else {
    std::cout << "WARNING: Class size exceeds maximum capacity." << std::endl;
  }
  return 0;
}
```

Using #define preprocessor directive (C-style)

```
#include <iostream>
#define a 5
int main() {
  std::cout<<a;
  return 0;
}
```

Using constexpr keyword (C++11 and later):

```cpp
#include <iostream>
constexpr int fibonacci(int n) {
  if (n <= 1) {
    return n;
  } else {
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
int main() {
  constexpr int nthTerm = 6;
  int result = fibonacci(nthTerm);
  std::cout << nthTerm << "th term of Fibonacci sequence: " << result << std::endl;
  return 0;
}
```

8). What are manipulators? How can we use them (Show it by program)

In C++, manipulators are special functions or operators that you use with input/output streams (cin, cout, etc.) to format the output or modify the way data is displayed or extracted. They are not functions in the traditional sense but rather helper functions that work with streams.

Here are some key points about manipulators:

- **Formatting Output:** Manipulators allow us to control how data is displayed on the screen, including setting field width, precision for floating-point numbers, alignment, and more.
- **Modifying Input/Output Behavior:** Some manipulators can control the behavior of input and output streams, such as enabling/disabling buffering or setting the base for numeric conversions.
- **Used with Insertion (<<) and Extraction (>>) Operators:** Manipulators are typically used in conjunction with the insertion (<<) operator for output streams and the extraction (>>) operator for input streams.

Here are some common examples of manipulators in C++:

- endl: Inserts a newline character and flushes the output buffer.
- setw(n): Sets the minimum field width for the next element to be printed (n spaces).
- setprecision(n): Sets the number of digits displayed after the decimal point for floating-point numbers.
- fixed: Forces fixed-point notation for floating-point numbers.
- scientific: Forces scientific notation for floating-point numbers.
- hex: Sets the base for numeric output to hexadecimal (base 16).
- dec: Sets the base for numeric output to decimal (base 10).
- showpos: Forces a plus sign to be displayed for positive numbers.
- noshowpos: Prevents a plus sign from being displayed for positive numbers.

By using manipulators effectively, one can make program's output more readable, consistent, and user-friendly.Here's an example:

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
  int age = 30;
  double pi = 3.14159;
  // Set field width and precision for formatted output
  cout << "Age: " << setw(5) << age << endl;
  cout << "Pi (2 decimal places): " << fixed << setprecision(2) << pi << endl;
  return 0;
}
```

9). What is dynamic memory allocation(DMA)? How can we implement this? Write a program to show DMA.

Dynamic memory allocation (DMA) is a technique in C++ that allows you to allocate memory for variables and objects at runtime, during program execution. This is in contrast to static memory allocation, where memory for variables is fixed at compile time based on their declaration.

C++ provides the new and delete operators for dynamic memory allocation and deallocation:

- **new operator:** This operator allocates a block of memory of a specific size (in bytes) on the heap (a region of memory managed by the program) and returns a pointer to the first byte of the allocated memory.
- **delete operator:** This operator deallocates the memory block pointed to by a pointer, releasing it back to the heap for potential reuse. It's crucial to deallocate memory when you're done using it to avoid memory leaks.

```cpp
#include <iostream>
using namespace std;
int main() {
  // Allocate memory for an integer at runtime
  int* age = new int;
  *age = 25; // Assign value to the allocated memory
  cout << "Age: " << *age << endl;
  // Deallocate memory after use
  delete age;
  return 0;
}
```

10). What is function overloading? Overload functions with

a) Different number of arguments b) Different types of arguments

In C++, function overloading is a feature that allows you to define multiple functions with the same name but different parameter lists. The compiler distinguishes between these functions based on the number, type, or order of their arguments. This can be useful for creating functions that perform similar operations on different data types or with different functionalities depending

on the provided arguments.

Overloading functions with different number of arguments

```cpp
#include <iostream>
using namespace std;
// Function to calculate the area of a square (takes one argument)
double calculateAreaSquare(double side) {
  return side * side;
}
// Function to calculate the area of a rectangle (takes two arguments)
double calculateAreaRectangle(double length, double width) {
  return length * width;
}
int main() {
  // Calculate area of square
  double squareArea = calculateAreaSquare(5.0);
  cout << "Area of square: " << squareArea << endl;
  // Calculate area of rectangle
  double rectArea = calculateAreaRectangle(4.0, 3.0);
  cout << "Area of rectangle: " << rectArea << endl;
  // Calculate area of triangle
  return 0;
}
```

Overloading functions with different types of arguments.

Overloading functions with different types of arguments.

```cpp
#include <iostream>
using namespace std;
void printVal(int A);
void printVal(char A);
void printVal(float A);
int main() {
  printVal(10);
  printVal('@');
  printVal(3.14f);
  return 0;
}
void printVal(int A)
{
  cout << endl << "Value of A : " << A;
}
void printVal(char A)
{
  cout << endl << "Value of A : " << A;
}
void printVal(float A)
{
  cout << endl << "Value of A : " << A;
}
```

11). What are inline functions?WAP to implement inline functions. What are the advantages of inline functions ? Where we cannot use inline functions?

In C++, inline functions are a compiler directive that suggests to the compiler to substitute the code within the function definition directly into the code of the calling function, rather than creating a separate set of instructions in memory. This can potentially improve performance by avoiding the overhead associated with function calls, such as pushing arguments onto the stack, jumping to the function body, and returning from the function.

```cpp
#include <iostream>

// Inline function definition
inline int add(int a, int b) {
    return a + b;
}

inline int multiply(int a, int b) {
    return a * b;
}

int main() {
    int x = 5, y = 10;

    // Calling the inline functions
    std::cout << "Addition of " << x << " and " << y << " is: " << add(x, y) << std::endl;
    std::cout << "Multiplication of " << x << " and " << y << " is: " << multiply(x, y) << std::endl;

    return 0;
}
```

The advantages of inline functions are:

Reduced Function Call Overhead: By inlining the function code, you can eliminate the overhead associated with function calls, potentially leading to performance gains for small and frequently called functions.

Improved Code Readability: Inlining can sometimes make code more readable by placing the function's implementation directly where it's used.

Inline functions can’t be used in the following places and domains:

1. Functions with Loops or Conditionals:

- The compiler typically won't inline functions containing loops (for, while, do-while) or complex conditional statements (if-else, switch). This is because inlining such functions could lead to code duplication and potential inefficiencies. Inlining the loop or conditional multiple times for each call site might outweigh any benefits from avoiding the function call overhead.

2. Recursive Functions:

- Inline functions with recursion can cause issues. Inlining a recursive function essentially copies the function body for each recursive call. This can lead to stack overflow errors if the recursion depth becomes too large. The compiler usually avoids inlining recursive functions for this reason.

3. Functions with Static Variables:

- Inline functions cannot have static variables. Static variables within a function are allocated memory only once during program execution, and they retain their values between function calls. Inlining the function with a static variable would create multiple copies of the static variable, leading to unexpected behavior.

4. Functions Returning Complex Data Structures:

- Inlining functions that return large or complex data structures (e.g., structures, classes) can significantly increase the code size. Copying the entire data structure for each call site might not be efficient. The compiler might choose not to inline such functions.

5. When Function Address is Taken:

- If you explicitly take the address of an inline function using the & operator, the compiler cannot inline it. This is because function pointers need to refer to the actual function code in memory, not the inlined version at the call site.

6. Compiler-Specific Restrictions:

- Different compilers might have their own limitations on inline functions. It's always

recommended to consult the compiler documentation for specific guidelines.

12). What are default arguments? What are the rules to define default arguments? Show with program.

In C++, default arguments are a feature that allows you to provide pre-defined values for function parameters in the function declaration. These default values are automatically used by the compiler when the function is called without providing arguments for those specific parameters.

Here are the rules for defining default arguments:

1. **Placement**: Default arguments must be specified from right to left. Once a parameter has a default value, all parameters to its right must also have default values.
2. **Declaration and Definition**: Default arguments are typically specified in the function declaration, not the definition. If provided in both, the values must match.
3. **Overloading**: Default arguments can be used with function overloading, but care must be taken to avoid ambiguity.
4. **Single Definition**: Default arguments should only be provided once, typically in the function declaration in a header file.

```cpp
#include <iostream>

// Function declaration with default arguments
void display(int a = 10, int b = 20);

int main() {
    // Calling the function without any arguments
    display();

    // Calling the function with one argument
    display(30);

    // Calling the function with both arguments
    display(40, 50);

    return 0;
}

// Function definition
void display(int a, int b) {
    std::cout << "a: " << a << ", b: " << b << std::endl;
}
```

13). What is pass by value and pass by reference in function call? Show with program.

Pass by Value

In pass by value, a copy of the actual argument's value is passed to the function. Any changes made to the parameter within the function do not affect the original variable in the calling function.

```cpp
#include <iostream>
void swapValues(int a, int b) { // Pass by value
  int temp = a;
  a = b;
  b = temp;
}
int main() {
  int x = 5, y = 10;
  std::cout << "Before swap: x = " << x << ", y = " << y << std::endl;
  swapValues(x, y); // Copies of x and y are passed
  std::cout << "After swap (in main): x = " << x << ", y = " << y << std::endl;
  return 0;
}
```

Pass by Reference

In pass by reference, the address (memory location) of the actual argument is passed to the function. Any changes made to the parameter within the function directly modify the original variable in the calling function.

```cpp
#include <iostream>
void swapValues(int& a, int& b) { // Pass by reference
  int temp = a;
  a = b;
  b = temp;
}
int main() {
  int x = 5, y = 10;
  std::cout << "Before swap: x = " << x << ", y = " << y << std::endl;
  swapValues(x, y); // Addresses of x and y are passed
  std::cout << "After swap (in main): x = " << x << ", y = " << y << std::endl;
  return 0;
}
```

14). What is the reference variable? Show return by reference with example.

A reference variable is essentially an alias or another name for an already existing variable. It doesn't create a new storage location; instead, it points to the same memory location as the original variable.

```cpp
#include <iostream>
int& get_value(int& x) {
    return x;
}
int main() {
    int num = 10;
    int& ref = get_value(num);
    std::cout << "Value of num: " << num << std::endl;  // Output: 10
    std::cout << "Value of ref: " << ref << std::endl;   // Output: 10
    ref = 20;  // Modify the value through the reference
    std::cout << "Value of num after modification: " << num << std::endl;  // Output: 20
    std::cout << "Value of ref after modification: " << ref << std::endl;   // Output: 20
    return 0;
}
```

15). Differentiate between

C++ structure and Class

Feature	Structure	Class
Default access specifier	Public	Private
Encapsulation	Low	High
Inheritance	Not supported	Supported
Polymorphism	Not supported	Supported
Constructors and destructors	Not supported	Supported
Member functions	Can have, but less common	Commonly used
Use cases	Simple data grouping	Complex data structures and object-oriented programming

C++ Macros and Inline Functions

Feature	Macros	Inline Functions
Definition	`#define`	`inline` keyword
Processing	Preprocessor	Compiler
Type checking	No	Yes
Side effects	Potential	Less likely
Debugging	Difficult	Easier
Efficiency	Can be faster, but less predictable	Often faster, but depends on compiler

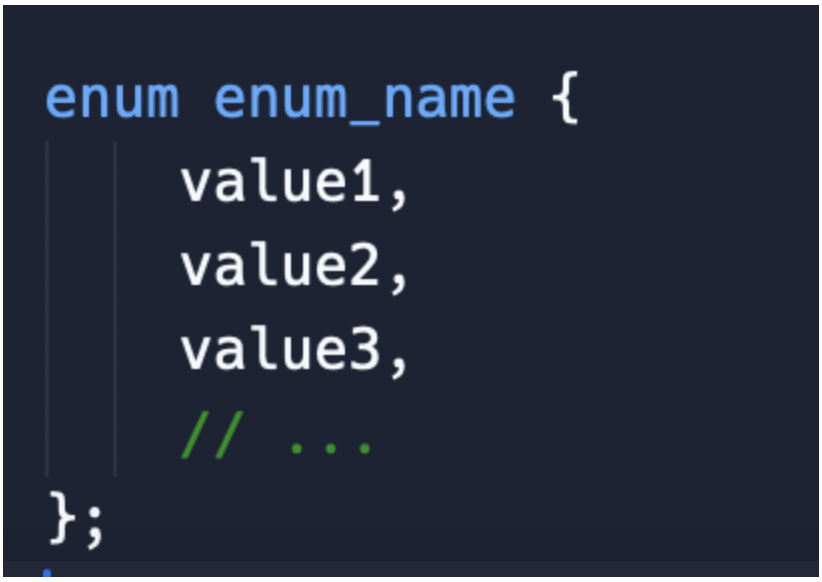
C++ structure and union

Feature	Structure	Union
Memory Allocation	Separate for each member	Shared by all members
Data Storage	Multiple members can hold values simultaneously	Only one member can hold a value at a time
Size	Sum of member sizes	Size of largest member
Usage	To group related data	To store different types of data in the same memory location

16). What are enumerations in C++? WAP to show the use of enumerations in C++.

Enumerations (enums) are user-defined data types in C++ that consist of a set of named integer constants. They provide a way to assign meaningful names to integral values, making code more readable and maintainable.

Syntax of enum:



Program to show the use of enumerations in C++

```cpp
#include <iostream>
enum DaysOfWeek {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
};
int main() {
    DaysOfWeek today = Wednesday;
    // Print the value of today
    std::cout << "Today is " << today << std::endl;
    // Accessing enum constants
    std::cout << "The value of Monday is " << Monday << std::endl;
    // Using enum in a switch statement
    switch (today) {
        case Monday:
            std::cout << "It's Monday, time to start the week!" << std::endl;
            break;
        case Friday:
            std::cout << "It's Friday, weekend is near!" << std::endl;
            break;
        case Saturday:
        case Sunday:
            std::cout << "It's weekend time!" << std::endl;
            break;
        default:
            std::cout << "It's a weekday." << std::endl;
    }
    return 0;
}
```

17). WAP to calculate simple interest by using the concept of default arguments with default values of rate = 15%.

```cpp
#include <iostream>
using namespace std;
int interest(int p,int t, int r = 15)
{
  return (p*t*r)/100;
}
int main()
{
  cout<<"The interest with default argument(p=1000,t=2): "<<interest(1000,2)<<endl;
  cout<<"The interest without default argument(p=1000,t=2)"<<interest(1000,2,13);
  return 0;
}
```

18). WAP to calculate the area of circle, triangle and rectangle using the concept of function overloading.

```cpp
#include <iostream>
using namespace std;
float area(int r)
{
  return 3.14*r*r;
}
float area(float x,int b, int h)
{
  return x * b* h;
}
int area(int l,int b)
{
  return l*b;
}
int main()
{
  cout<<"Area of triangle(base =3,height =4): "<<area(0.5,3,4)<<endl;
  cout<<"Area of rectangle(length =3,breath =4): "<<area(3,4)<<endl;
  cout<<"Area of circle(radius = 3): "<<area(3)<<endl;
  return 0;
}
```

19). WAP to find the volume of cube, cuboid and cylinder using the concept of function overloading.

```cpp
#include <iostream>
using namespace std;
float vol(int r, int h)
{
  return 3.14 * r * r * h;
}
float vol(int l)
{
  return l*l*l;
}
float vol(int l, int b, int h)
{
  return l*b*h;
}
int main()
{
  cout<<"Volume of cube( length = 3): "<<vol(3)<<endl;
  cout<<"Volume of cylinder(radius = 3, height 4): "<<vol(3,4)<<endl;
  cout<<"Volume of cuboid( length = 3, breadth = 4, height = 5): "<<vol(3,4,5)<<endl;
  return 0;
}
```

20). Write a program to display N numbers of characters by using default arguments for both parameters. Assume that the function takes two arguments: one character to be printed and another number of characters to be printed.

```cpp
#include <iostream>
using namespace std;
int display(char c = 'p', int n = 5)
{
  for(int i=0;i<5;i++)
    {
      cout<<c<<endl;
    }
  return 0;
}
int main()
{
  cout<<"Using default arguments:";
  display();
  cout<<"Without using default arguments:";
  display('m',8);
  return 0;
}
```