Unit 1 (Introduction to C++)

Procedure Programming

1. A procedure is a self-contained block of statements that performs a specific task within a program.
2. Procedures allow a program to be broken down into smaller, manageable pieces or modules. Each procedure performs a specific function and can be developed independently.
3. Procedural programs follow a sequence of instructions. The flow of the program moves from one statement to the next, in the order they appear in the code.
4. Variables declared within a procedure, accessible only within that procedure.
5. Variables declared outside of all procedures, accessible from any procedure within the program.
6. Procedures can accept inputs, known as parameters, which can be used to provide data or influence the procedure's execution.
7. Procedures can also return outputs, which can be used by other parts of the program.
8. Procedural programming often involves structured programming principles, such as using loops, conditionals, and other control structures to manage the flow of execution within and between procedures.

Advantages of Procedural Programming

- **Simplicity**: The linear and straightforward approach of procedural programming makes it easy to understand and implement.
- **Modularity**: Breaking down a program into smaller procedures makes it easier to manage, debug, and test.
- **Reusability**: Procedures can be reused across different parts of a program, reducing code duplication.
- **Maintainability**: Well-structured procedures make the program easier to maintain and modify over time.

Disadvantages of Procedural Programming

- **Scalability**: For very large programs, procedural programming can become complex and difficult to manage.
- **Data Hiding**: Procedural programming doesn't inherently support data hiding, making it easier for data to be accidentally modified.
- **Object-Oriented Features**: Procedural programming lacks the features of object-oriented programming, such as inheritance and polymorphism, which can be useful for certain types of applications.

Object-oriented programming

1. Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs.
2. It utilizes several key concepts, including inheritance, encapsulation, polymorphism, and abstraction.
3. Objects are instances of classes and represent entities that have attributes (data) and behaviors (methods or functions).
4. The state of an object is defined by its attributes, and its behavior is defined by its methods.
5. A class is a blueprint or template for creating objects. It defines the attributes and methods that the objects created from the class will have.
6. Code is organized into classes and objects, which makes it modular and easier to manage.
7. Inheritance and polymorphism promote the reuse of existing code, which reduces redundancy.
8. OOP systems are easier to scale as they can be extended with new classes and objects without modifying existing code.

Advantages of Object-Oriented Programming

- **Modularity**: Code is organized into classes and objects, which makes it modular and easier to manage.
- **Reusability**: Inheritance and polymorphism promote the reuse of existing code, which reduces redundancy.

- **Maintainability**: Encapsulation ensures that objects control their own state, making the system more robust and easier to maintain.
- **Scalability**: OOP systems are easier to scale as they can be extended with new classes and objects without modifying existing code.
- **Flexibility**: Polymorphism and abstraction make it easier to implement systems that can handle a variety of situations.

Disadvantages of Object-Oriented Programming

- **Complexity**: OOP can be more complex to design and understand compared to procedural programming, especially for smaller programs.
- **Performance**: The abstraction layers in OOP can introduce overhead, making it less efficient in terms of performance for certain applications.
- **Learning Curve**: OOP concepts like inheritance, polymorphism, and encapsulation can be challenging for beginners to grasp.

Difference in OOP and Procedural Programming

Procedural Oriented Programming	Object-O
In procedural programming, the program is divided into small parts called ***functions***.	In object-oriente divided into
Procedural programming follows a ***top-down approach***.	Object-oriented p
There is no access specifier in procedural programming.	Object-oriented p like priva
Adding new data and functions is not easy.	Adding new
Procedural programming does not have any proper way of hiding data so it is ***less secure***.	Object-oriented p so
In procedural programming, overloading is not possible.	Overloading
In procedural programming, there is no concept of data hiding and inheritance.	In object-oriente data hidin
In procedural programming, the function is more important than the data.	In object-orient impo
Procedural programming is based on the ***unreal world***.	Object-oriented p

Applications of OOP

1. Desktop Applications: OOP is used in creating robust and user-friendly desktop applications, including word processors, spreadsheets, and media players. Examples include Microsoft Office and Adobe Photoshop.

2. Web Applications: Modern web applications use OOP principles to manage and organize code, making it easier to develop, maintain, and scale. Frameworks like Django (Python), Ruby on Rails (Ruby), and Laravel (PHP) are based on OOP concepts.

3. Game Engines: OOP is extensively used in game development, where objects represent characters, enemies, weapons, and other game elements. Popular game engines like Unity and Unreal Engine are built on OOP principles.

4. Robotics: OOP helps in designing robotic systems where different components (sensors, actuators, control units) are modeled as objects.

5. Android and iOS Apps: Mobile app development heavily relies on OOP. Frameworks like Android's SDK (Java/Kotlin) and iOS's Swift provide OOP tools and libraries to build feature-rich mobile applications.

Introduction to C++

1. C++ is a high-level programming language.
2. It is an extension of the C programming language and incorporates object-oriented programming features, making it a powerful tool for software development.
3. C++ supports the four fundamental OOP principles: encapsulation, inheritance, polymorphism, and abstraction.
4. C++ provides low-level memory manipulation features, such as pointers and manual memory management, which offer high performance and control over system resources.
5. C++ maintains a high degree of compatibility with C, making it easy for C programmers to transition to C++ and leverage existing C libraries.
6. C++ is designed to be a high-performance language. It compiles to native machine code, which makes it suitable for resource-intensive applications such as game development, real-time systems, and large-scale simulations.
7. C++ supports multiple programming paradigms, including procedural, object-oriented, and generic programming. This flexibility allows developers to choose the best approach for their specific needs.

Structure of C++ Program

1. Preprocessor Directives

Preprocessor directives are lines included in the code of programs preceded by a hash symbol (#). These lines are processed by the preprocessor before the actual compilation begins.

2. Main Function

Every C++ program must have a main function, which is the entry point of the program. Execution of a C++ program starts from the main function.

3. Variable Declarations

Variables are used to store data. They must be declared before they are used in the program.

4. Statements and Expressions

Statements are the instructions given to the computer to perform any kind of action, such as variable assignments, loops, conditionals, etc.

5. Functions

Functions are blocks of code that perform specific tasks and can be called from the main function or other functions.

6. Classes and Objects

Classes are user-defined data types that represent real-world entities. Objects are instances of classes.

Example:

```
#include <iostream>

int add(int x, int y) {
    return x + y;
}

class Rectangle {
public:
    int width, height;
    int area() {
        return width * height;
    }
};

int main() {
    int num1 = 3, num2 = 7;
```

```
    std::cout << "Sum: " << add(num1, num2) << std::endl;

    Rectangle rect;
    rect.width = 5;
    rect.height = 10;
    std::cout << "Area: " << rect.area() << std::endl;

    return 0;
}
```

Keywords

Keywords are reserved words in C++ that have special meanings and cannot be used for other purposes such as variable names or function names. They are part of the language syntax. Some examples of C++ keywords are:

- int
- float
- double
- char
- void
- if
- else
- switch
- case
- for
- while

Identifiers

Identifiers are names given to various program elements such as variables, functions, arrays, classes, etc. They must follow certain rules:

- Can contain letters (both uppercase and lowercase), digits, and underscores (_).
- Must begin with a letter or an underscore.
- Cannot be a keyword.

```
myVariable, _myVariable, my_variable1, MyClass
```

Constants

Constants are fixed values that do not change during the execution of a program. They can be of various types:

- **Integer Constants**: Whole numbers without a decimal point. Example: `10`, `-20`.
- **Floating-Point Constants**: Numbers with a decimal point. Example: `3.14`, `-0.001`.
- **Character Constants**: Single characters enclosed in single quotes. Example: `'A'`, `'9'`.
- **String Constants**: A sequence of characters enclosed in double quotes. Example: `"Hello, World!"`.
- **Boolean Constants**: Represent true or false values. Example: `true`, `false`.

```
const int MAX = 100;
const float PI = 3.14159;
const char NEWLINE = '\n';
```

Data Types

Data types specify the type of data that a variable can hold. In C++, data types are categorized into several groups:

1. Basic Data Types:

-
  - `int`: Integer type.
  - `char`: Character type.
  - `float`: Single-precision floating-point type.
  - `double`: Double-precision floating-point type.

    - `bool`: Boolean type (true or false).
    - `void`: Represents the absence of type.

```
int age = 25;
char grade = 'A';
float temperature = 36.6;
double pi = 3.1415926535;
bool isTrue = true;
```

2. Derived Data Types:

- **Arrays**: Collection of elements of the same type.
- **Pointers**: Variables that store memory addresses.
- **References**: Alias for another variable.
- **Function**: Functions returning a value.

```
int arr[10];
int* ptr;
int& ref = age;
```

3. User-Defined Data Types:

- **Structures (`struct`)**: Group of variables of different types.
- **Unions (`union`)**: Similar to structures but with shared memory for all members.
- **Enumerations (`enum`)**: Set of named integer constants.
- **Classes**: Blueprint for creating objects.

```
struct Person {
    char name[50];
    int age;
};

union Data {
    int intVal;
    float floatVal;
};

enum Color {
    RED, GREEN, BLUE
};

class Car {
    public:
    string brand;
```

```
    int year;
};
```

Types of Operators

Arithmetic Operators

Used for mathematical operations:

- `+` (Addition): `a + b`
- `-` (Subtraction): `a - b`
- `*` (Multiplication): `a * b`
- `/` (Division): `a / b`
- `%` (Modulus): `a % b`

Relational Operators

Used to compare two values:

- `==` (Equal to): `a == b`
- `!=` (Not equal to): `a != b`
- `>` (Greater than): `a > b`
- `<` (Less than): `a < b`
- `>=` (Greater than or equal to): `a >= b`
- `<=` (Less than or equal to): `a <= b`

Logical Operators

Used to combine multiple conditions:

- `&&` (Logical AND): `a && b`
- `||` (Logical OR): `a || b`
- `!` (Logical NOT): `!a`

Bitwise Operators

Used for bit-level operations:

- `&` (Bitwise AND): `a & b`

- `|` (Bitwise OR): `a | b`
- `^` (Bitwise XOR): `a ^ b`
- `~` (Bitwise NOT): `~a`
- `<<` (Left shift): `a << 2`
- `>>` (Right shift): `a >> 2`

Assignment Operators

Used to assign values to variables:

- `=` (Assignment): `a = b`
- `+=` (Add and assign): `a += b`
- `-=` (Subtract and assign): `a -= b`
- `*=` (Multiply and assign): `a *= b`
- `/=` (Divide and assign): `a /= b`
- `%=` (Modulus and assign): `a %= b`
- `&=` (Bitwise AND and assign): `a &= b`
- `|=` (Bitwise OR and assign): `a |= b`
- `^=` (Bitwise XOR and assign): `a ^= b`
- `<<=` (Left shift and assign): `a <<= b`
- `>>=` (Right shift and assign): `a >>= b`

Increment and Decrement Operators

Used to increase or decrease the value of a variable by one:

- `++` (Increment): `a++` or `++a`
- `--` (Decrement): `a--` or `--a`

Conditional (Ternary) Operator

A shorthand for if-else statements:

- `?:` (Ternary): `condition ? expression1 : expression2`

Comma Operator

Used to separate two or more expressions:

- `,` (Comma): `a = (b = 3, b + 2)`

Unit 2 (Classes, Objects & Functions in C++)

Function

1. Functions in Programming is a block of code that encapsulates a specific task or related group of tasks.
2. Functions are defined by a name, may have parameters and may return a value.
3. The main idea behind functions is to take a large program, break it into smaller, more manageable pieces (or functions), each of which accomplishes a specific task.
4. Functions in Programming allow programmers to abstract the details of a particular operation.
5. Instead of dealing with the entire implementation, a programmer can use a function with a clear interface, relying on its functionality without needing to understand the internal complexities.
6. Well-designed functions enhance code readability by providing a clear structure and isolating specific tasks.
7. This makes it easier for programmers to understand and maintain the code, especially in larger projects where complexity can be a challenge.

Function example

```
#include <iostream>

// Function declaration
int add(int a, int b);

int main() {
    int result = add(5, 3);  // Calling the function
    std::cout << "The result is: " << result << std::endl;
    return 0;
}

// Function definition
int add(int a, int b) {
    // Function body
    return a + b;
}
```

- **Function Declaration:** `int add(int a, int b);` - This declares the function `add` which takes two integers as parameters and returns an integer.

- **Function Call:** `add(5, 3);` - This calls the function `add` with arguments `5` and `3`.

- The function body is the part between the `{` and `}` braces, containing the code that specifies what the function does. In this case, it adds the two integers `a` and `b` and returns the result.

No argument function

A no-argument function in C++ is a function that does not take any parameters.

```
#include <iostream>

// Function declaration
void greet();

int main() {
    greet();  // Calling the no-argument function
    return 0;
}

// Function definition
void greet() {
    // Function body
    std::cout << "Hello, World!" << std::endl;
}
```

- **Function Declaration:** `void greet();` - This declares the function `greet` which takes no parameters and returns no value (`void`).

- **Function Call:** `greet();` - This calls the function `greet`.

Recursive function

A recursion function in C++ is a function that calls itself in order to solve a problem. Recursion is often used to solve problems that can be broken down into smaller, simpler subproblems of the same type.

```
#include <iostream>

// Function declaration
int factorial(int n);

int main() {
    int number = 5;
    std::cout << "Factorial of " << number << " is " <<
factorial(number) << std::endl;
    return 0;
}

// Function definition
int factorial(int n) {
    if (n <= 1) {
        return 1;  // Base case
    } else {
        return n * factorial(n - 1);  // Recursive case
    }
}
```

- **Function Declaration:** `int factorial(int n);` - This declares the function `factorial` which takes an integer `n` as a parameter and returns an integer.

- **Function Call:** `factorial(number);` - This calls the function `factorial` with the argument `number`.

- In the function definition, the function calls itself in the recursive case `return n * factorial(n - 1);` until it reaches the base case `if (n <= 1)`, which stops the recursion.

- The base case is necessary to prevent infinite recursion and to provide a simple solution to the smallest subproblem.

Function overloading

1. Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters.

2. When a function name is overloaded with different jobs it is called Function Overloading.

3. In Function Overloading “Function” name should be the same and the arguments should be different.

4. Function overloading can be considered as an example of a polymorphism feature in C++.

5. If multiple functions having same name but parameters of the functions should be different is known as Function Overloading.

6. If we have to perform only one operation and having same name of the functions increases the readability of the program.

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

void add(int a, int b)
{
cout << "sum = " << (a + b);
}

void add(double a, double b)
{
cout << endl << "sum = " << (a + b);
}

int main()
{
add(10, 2);
add(5.3, 6.2);

return 0;
}
```

Classes and Objects

1. A class in C++ is a blueprint for creating objects. It defines a type that groups data and functions together into a single unit.

2. The data members (or attributes) hold information about the state of an object, while member functions (or methods) define the behavior of the object.

3. Classes encapsulate data and functions that operate on the data, restricting direct access to some of the object's components and ensuring that data integrity is maintained.

4. An object is an instance of a class.

5. When a class is defined, no memory is allocated until an object of that class is created.

6, Each object has its own set of data members and can use member functions to interact with or modify these data members.

Example of implementing class

```
// Create a Car class with some attributes
class Car {
  public:
    string brand;
    string model;
    int year;
};

int main() {
  // Create an object of Car
  Car carObj1;
  carObj1.brand = "BMW";
  carObj1.model = "X5";
  carObj1.year = 1999;

  // Create another object of Car
  Car carObj2;
  carObj2.brand = "Ford";
  carObj2.model = "Mustang";
  carObj2.year = 1969;

  // Print attribute values
  cout << carObj1.brand << " " << carObj1.model << " " <<
carObj1.year << "\n";
```

```
  cout << carObj2.brand << " " << carObj2.model << " " <<
carObj2.year << "\n";
  return 0;
}
```

Public access modifier

1. All the class members declared under public will be available to everyone.
2. The data members and member functions declared public can be accessed by other classes too.
3. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

```
// C++ program to demonstrate public
// access modifier

#include <iostream>
using namespace std;

// class definition
class Circle {
public:
double radius;

double compute_area()
{
return 3.14 * radius * radius;
}
};

// main function
int main()
{
Circle obj;

// accessing public data member outside class
obj.radius = 5.5;

cout << "Radius is: " << obj.radius << "\n";
cout << "Area is: " << obj.compute_area();
return 0;
```

```
}
```

Private access modifier

1. The class members declared as private can be accessed only by the functions inside the class.
2. Only the member functions or the friend functions are allowed to access the private data members of a class.
3. They are not allowed to be accessed directly by any object or function outside the class.

```
// C++ program to demonstrate private
// access modifier

#include <iostream>
using namespace std;

class Circle {
// private data member
private:
double radius;

// public member function
public:
void compute_area(double r)
{
// member function can access private
// data member radius
radius = r;

double area = 3.14 * radius * radius;

cout << "Radius is: " << radius << endl;
cout << "Area is: " << area;
}
};

// main function
int main()
{
// creating object of the class
```

```
Circle obj;

// trying to access private data member
// directly outside the class
obj.compute_area(1.5);

return 0;
}
```

Unit 3 (Constructors, Destructors & Operator Overloading)

Constructor

1. Constructor is a member function of a class, whose name is the same as the class name.
2. Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically when an object of the same class is created.
3. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as a constructor.
4. Constructors do not return value, hence they do not have a return type.
5. A constructor gets called automatically when we create the object of the class.
6. Constructors can be overloaded.
7. A constructor can not be declared virtual.

Parameterized constructors

Parameterized constructors in C++ are special types of constructors that allow you to initialize objects with specific values when they are created. These constructors take one or more parameters that are used to set the initial values of the object's data members. This is useful for creating objects with different initial states.

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

class Rectangle {
private:
    int width;
    int height;

public:
    // Parameterized constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    // Function to calculate the area
    int area() {
        return width * height;
    }
```

```
    // Function to display the dimensions
    void display() {
        cout << "Width: " << width << ", Height: " << height << endl;
    }
};

int main() {
    // Creating objects using the parameterized constructor
    Rectangle rect1(10, 20);
    Rectangle rect2(5, 15);

    // Displaying the dimensions and area of the rectangles
    rect1.display();
    cout << "Area: " << rect1.area() << endl;

    rect2.display();
    cout << "Area: " << rect2.area() << endl;

    return 0;
}
```

In this example:

- The `Rectangle` class has a parameterized constructor that takes two parameters, `w` and `h`, which are used to initialize the `width` and `height` data members.
- When we create objects `rect1` and `rect2`, we pass specific values for `w` and `h`, which set the initial dimensions of these rectangle objects.
- The `area` method calculates the area of the rectangle, and the `display` method prints the width and height.

Default argument constructor

In C++, constructors can also have default arguments, allowing you to create objects with or without specifying all parameters. When a parameter is not provided, the default value is used.

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

class Rectangle {
private:
```

```
    int width;
    int height;

public:
    // Constructor with default arguments
    Rectangle(int w = 10, int h = 5) {
        width = w;
        height = h;
    }

    // Function to calculate the area
    int area() {
        return width * height;
    }

    // Function to display the dimensions
    void display() {
        cout << "Width: " << width << ", Height: " << height << endl;
    }
};

int main() {
    // Creating objects using the constructor with default arguments
    Rectangle rect1; // Uses default values for both width and height
    Rectangle rect2(20); // Uses default value for height
    Rectangle rect3(15, 25); // Uses provided values for both width
and height

    // Displaying the dimensions and area of the rectangles
    rect1.display();
    cout << "Area: " << rect1.area() << endl;

    rect2.display();
    cout << "Area: " << rect2.area() << endl;

    rect3.display();
    cout << "Area: " << rect3.area() << endl;

    return 0;
}
```

In this example:

- The `Rectangle` class has a constructor with default arguments, `int w = 10` and `int h = 5`.
- When we create the object `rect1`, no arguments are provided, so it uses the default values (width = 10, height = 5).

- When we create the object `rect2`, only one argument (20) is provided, so it uses the default value for height (height = 5).
- When we create the object `rect3`, both arguments (15 and 25) are provided, so it uses these values.

---

constructor overloading

1. In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments.

2. This concept is known as Constructor Overloading and is quite similar to function overloading.

3. Overloaded constructors essentially have the same name (exact name of the class) and different by number and type of arguments.

4. A constructor is called depending upon the number and type of arguments passed.

5. While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```
// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;

class construct
{

public:
float area;

// Constructor with no parameters
construct()
{
area = 0;
}

// Constructor with two parameters
construct(int a, int b)
{
area = a * b;
}
```

```
void disp()
{
cout<< area<< endl;
}
};

int main()
{
// Constructor Overloading
// with two different constructors
// of class name
construct o;
construct o2( 10, 20);

o.disp();
o2.disp();
return 1;
}
```

Dynamic initialization of objects

Dynamic initialization of objects in C++ involves creating objects at runtime using the `new` operator. This approach is particularly useful when the size or number of objects is not known at compile time. The `new` operator allocates memory for the object and calls the constructor to initialize it.

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

class Rectangle {
private:
    int width;
    int height;

public:
    // Parameterized constructor
    Rectangle(int w, int h) {
        width = w;
        height = h;
```

```
    }

    // Function to calculate the area
    int area() {
        return width * height;
    }

    // Function to display the dimensions
    void display() {
        cout << "Width: " << width << ", Height: " << height << endl;
    }

    // Destructor
    ~Rectangle() {
        cout << "Rectangle destroyed" << endl;
    }
};

int main() {
    // Dynamic initialization of objects
    Rectangle* rect1 = new Rectangle(10, 20);
    Rectangle* rect2 = new Rectangle(5, 15);

    // Displaying the dimensions and area of the rectangles
    rect1->display();
    cout << "Area: " << rect1->area() << endl;

    rect2->display();
    cout << "Area: " << rect2->area() << endl;

    // Deleting the dynamically allocated objects to free memory
    delete rect1;
    delete rect2;

    return 0;
}
```

---

copy constructor

1. A copy constructor is a member function that initializes an object using another object of the same class.

2. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a copy constructor.

3. Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.

4. Copy constructor takes a reference to an object of the same class as an argument.

5. The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

```
#include <iostream>
#include <string.h>
using namespace std;
class student {
int rno;
char name[50];
double fee;

public:
student(int, char[], double);
student(student& t) // copy constructor
{
rno = t.rno;
strcpy(name, t.name);
fee = t.fee;
}
void display();
};

student::student(int no, char n[], double f)
{
rno = no;
strcpy(name, n);
fee = f;
}

void student::display()
{
cout << endl << rno << "\t" << name << "\t" << fee;
}

int main()
{
```

```
student s(1001, "Manjeet", 10000);
s.display();

student manjeet(s); // copy constructor called
manjeet.display();

return 0;
}
```

Destructor

1. Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed.

2. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

3. Destructor has the same name as their class name preceded by a tilde (~) symbol.

4. It is not possible to define more than one destructor.

5. The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded.

6. Destructor neither requires any argument nor returns any value.

7. It is automatically called when an object goes out of scope.

8. Destructor release memory space occupied by the objects created by the constructor.

9. In destructor, objects are destroyed in the reverse of an object creation.

Destructor Example

The below code demonstrates the automatic execution of constructors and destructors when objects are created and destroyed, respectively.

```
// C++ program to demonstrate the execution of constructor
// and destructor

#include <iostream>
using namespace std;

class Test {
public:
// User-Defined Constructor
Test() { cout << "\n Constructor executed"; }

// User-Defined Destructor
~Test() { cout << "\nDestructor executed"; }
};
main()
{
Test t;

return 0;
}
```

Operator overloading

1. Operator overloading in C++ allows you to redefine the behavior of operators for user-defined types (such as classes and structures).

2. This feature enables objects of these types to interact using operators in a way that is intuitive and consistent with their intended use.

3. Operator overloading makes it possible to use operators with user-defined types in a manner similar to how they are used with built-in types.

4. For example, you can overload the + operator to add two objects of a custom class, or overload the << operator to output an object to a stream.

5. Operator overloading is achieved by defining a special member function or a friend function in the class.

6. The function's name is operator followed by the operator symbol being overloaded.

Rules for Operator Overloading

1. **Cannot Change Operator Precedence**

- You cannot change the precedence or associativity of operators. The precedence and associativity of an overloaded operator are the same as for the built-in operators.

2. **Cannot Create New Operators**

- You cannot create new operators. You can only overload existing operators.

3. **Cannot Change Operator Arity**

- You cannot change the number of operands that an operator takes. For example, you cannot make the binary `+` operator unary or the unary `-` operator binary.

5. **Overloading Must Be Done Within a Class or as a Friend Function**

- Operators must be overloaded either as member functions or as friend functions.
- Unary operators should typically be overloaded as member functions.
- Binary operators can be overloaded as either member functions or friend functions, depending on the context.

Type Conversion

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. Implicit Type Conversion

- Implicit Type Conversion Also known as ‘automatic type conversion’.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.
- It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

```
// An example of implicit conversion

#include <iostream>
using namespace std;

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;

cout << "x = " << x << endl
<< "y = " << y << endl
<< "z = " << z << endl;

return 0;
}
```

2. Explicit Type Conversion

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.
In C++, it can be done by two ways:

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.
- **Conversion using Cast operator:** A Cast operator is an unary operator which forces one data type to be converted into another data type.
- C++ supports four types of casting:

1. *Static Cast*

2. Dynamic Cast
3. *Const Cast*
4. *Reinterpret Cast*

```
//example demonstrating the conversion of a float value to an int using
different casting methods:

#include <iostream>
using namespace std;

int main() {
    float f = 3.6;

    // C-style cast
    int i1 = (int)f;
    cout << "C-style cast: " << i1 << endl;

    // Function-style cast
    int i2 = int(f);
    cout << "Function-style cast: " << i2 << endl;

    // static_cast
    int i3 = static_cast<int>(f);
    cout << "static_cast: " << i3 << endl;

    return 0;
}
```

Explanation:

1. **C-Style Cast**:
   - Converts the float `3.6` to an integer `3` by truncating the decimal part.
   - This method is less safe and less explicit, making it harder to locate in large codebases and understand the intent.
2. **Function-Style Cast**:
   - Also converts the float `3.6` to an integer `3`.
   - Similar to the C-style cast in behavior but uses a different syntax.
3. `static_cast`:
   - Converts the float `3.6` to an integer `3`.
   - Provides better type safety and is more explicit, which makes the code easier to understand and maintain.

Unit 4 (Inheritance)

Inheritance

1. Inheritance in C++ is a fundamental concept in object-oriented programming (OOP) that allows you to create new classes (derived classes or child classes) based on existing classes (base classes or parent classes).
2. The derived class inherits the properties (attributes) and behaviors (methods) of the base class, allowing for code reuse and creating a hierarchy of classes.
3. Key points about inheritance in C++:

- Base Class: The class from which properties are inherited.
- Derived Class: The class that inherits properties from another class.
- Code Reusability: Avoids writing the same code multiple times.
- Hierarchical Relationships: Organizes classes in a parent-child relationship.
- Access Specifiers: Controls the visibility of inherited members (public, protected, private).

Single inheritance

1. Single inheritance in C++ is a mechanism where a derived class inherits properties and methods from a single base class.

2. This promotes code reuse, hierarchical classification, and extensibility.

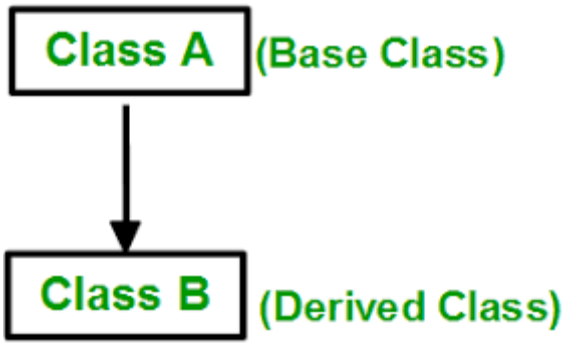
3. Inheritance is specified using the : symbol followed by an access specifier (public, protected, or private) and the base class name. 4. The access specifier controls how the base class members are accessible in the derived class.

5. Public inheritance retains the access levels, protected inheritance makes them protected, and private inheritance makes them private.

6. Constructors and destructors of the base class are called in a specific order during object creation and destruction but are not inherited.

7. Function overriding allows derived classes to provide specific implementations of base class methods, enabling polymorphism.

8. Single inheritance ensures a straightforward inheritance path, unlike multiple inheritance, which involves multiple base classes.




```
// C++ program to demonstrate how to implement the Single
// inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// sub class derived from a single base classes
class Car : public Vehicle {
public:
    Car() { cout << "This Vehicle is Car\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Multilevel inheritance

1. Multilevel inheritance in C++ is where a class is derived from another derived class, forming a chain.

2. It involves a base class, an intermediate class, and a final derived class. Each class in the chain inherits properties and methods from its predecessor.

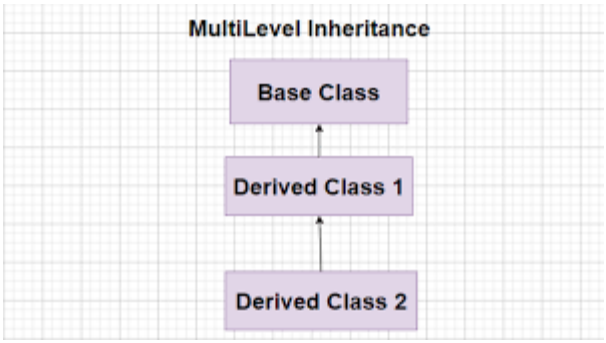
3. Inheritance is specified using the `:` symbol followed by an access specifier (public, protected, private) and the base class name.

4. This type of inheritance promotes extended code reuse across multiple levels, creating a clear hierarchical structure.

5. Constructors and destructors are called in order from the base class to the final derived class during object creation and in reverse during destruction.

6. Access specifiers determine the visibility of base class members in derived classes, similar to single inheritance.

7. Multilevel inheritance ensures a structured and extended inheritance path, enhancing code organization and reuse.




```
// C++ program to implement Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
```

```
class fourWheeler : public Vehicle {
public:
    fourWheeler() { cout << "4 Wheeler Vehicles\n"; }
};

// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
    Car() { cout << "This 4 Wheeler Vehical is a Car\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;
    return 0;
}
```

Multiple Inheritance

1. Multiple inheritance in C++ allows a class to inherit characteristics and behaviors from more than one base class.

2. This means a derived class can have multiple parent classes, enabling it to access and utilize the properties and methods from all of them.

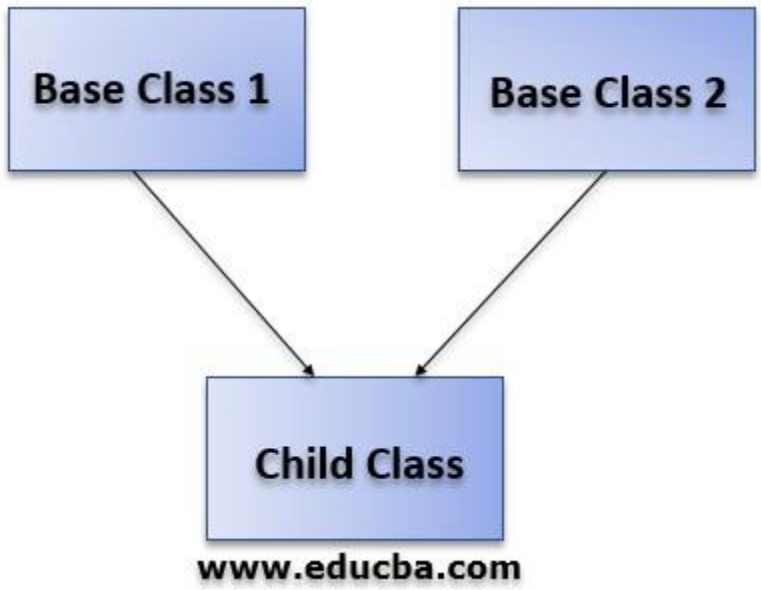
3. While this can be powerful for creating complex class relationships and reusable code, it can also introduce complications.

4. One such complication is the "diamond problem," where a class inherits from two classes that have a common base class.

5. This can lead to ambiguity and redundancy, as the derived class may inherit multiple copies of the base class's properties.

6. To manage these issues, C++ provides mechanisms like virtual inheritance, which ensures only one instance of the shared base class is inherited.

7. Despite its complexity, multiple inheritance can be useful in scenarios where a class needs to combine functionalities from diverse sources.




```
// C++ program to illustrate the multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// second base class
class FourWheeler {
public:
    FourWheeler() { cout << "This is a 4 Wheeler\n"; }
};

// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
  public:
    Car() { cout << "This 4 Wheeler Vehical is a Car\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
```

```
    Car obj;
    return 0;
}
```

Hierarchical inheritance

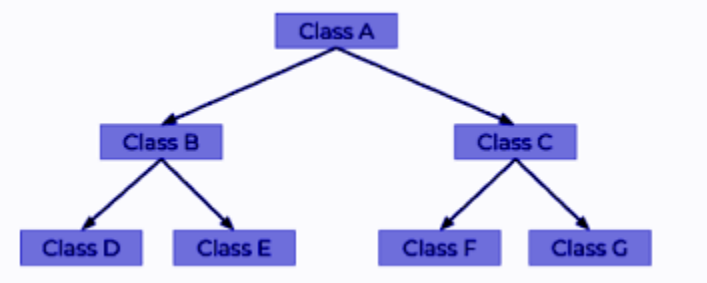
1. Hierarchical inheritance in C++ involves multiple derived classes inheriting from a single base class.

2. This form of inheritance creates a structure where one parent class serves as the foundation for multiple child classes.

3. Each derived class inherits the properties and behaviors of the base class, allowing them to share common functionality while also defining their unique features.

4. Hierarchical inheritance is particularly useful in scenarios where several subclasses need to implement the same interface or base functionality but also require specialized behaviors.

5. For example, in a graphical user interface framework, a base class for widgets might define common attributes like size and position, while derived classes for buttons, text boxes, and sliders inherit these attributes and add specific functionalities.




```
// C++ program to implement Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
```

```
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
public:
    Car() { cout << "This Vehicle is Car\n"; }
};

// second sub class
class Bus : public Vehicle {
public:
    Bus() { cout << "This Vehicle is Bus\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Car obj1;
    Bus obj2;
    return 0;
}
```

Hybrid inheritance

1. Hybrid inheritance in C++ combines two or more types of inheritance—such as single, multiple, hierarchical, and multilevel inheritance—within a single program.

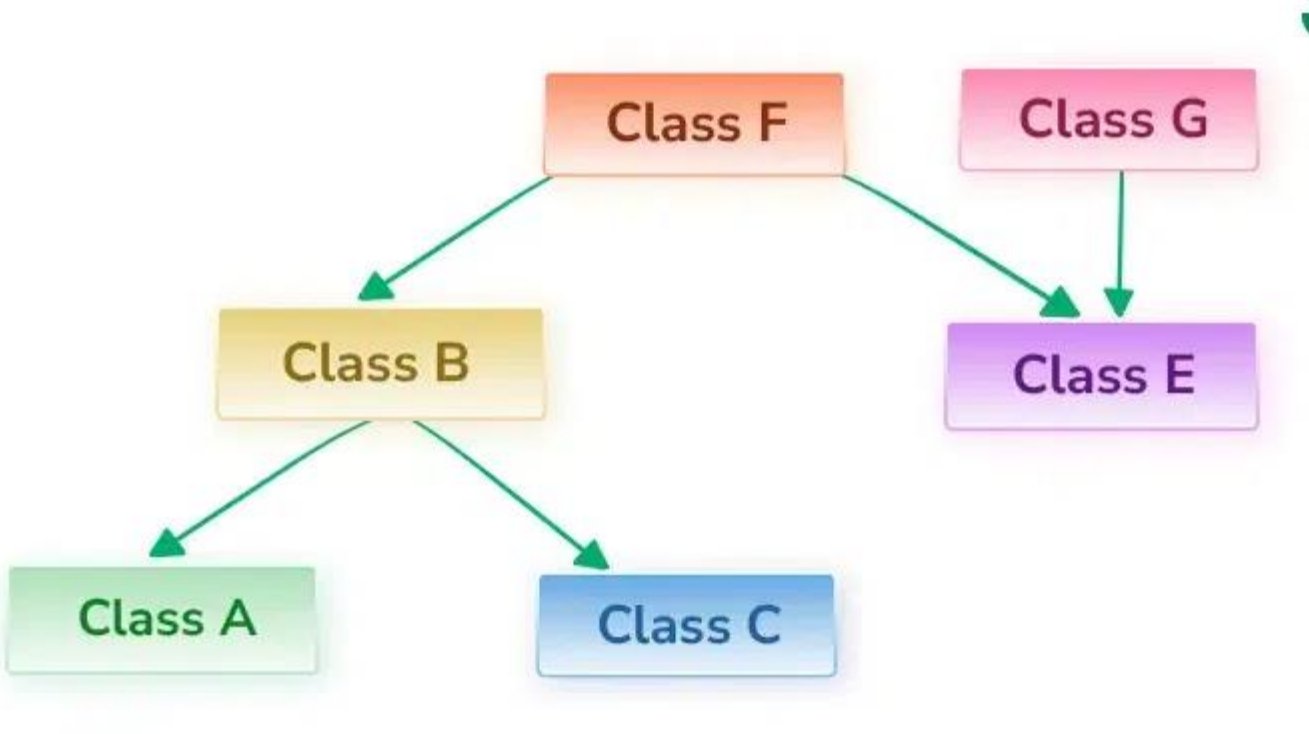
2. This results in a complex inheritance structure that leverages the strengths of various inheritance types to achieve a more flexible and powerful object-oriented design.

3. Hybrid inheritance can be used to model complex relationships by incorporating different inheritance strategies.

4. For example, a class might inherit from one class using single inheritance while also engaging in multiple inheritance with another class.

5. This allows for the creation of intricate class hierarchies that can handle diverse requirements.

6. However, hybrid inheritance can also introduce challenges, such as increased complexity and the potential for the "diamond problem" when multiple inheritance is involved.




```
// C++ program to illustrate the implementation of Hybrid Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// base class
class Fare {
public:
    Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
  public:
```

```
  Car() { cout << "This Vehical is a Car\n"; }
};

// second sub class
class Bus : public Vehicle, public Fare {
  public:
  Bus() { cout << "This Vehicle is a Bus with Fare\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Bus obj2;
    return 0;
}
```

Making Private Member Inheritable

```
// C++ program to illustrate how to access the private
// members of the base class by declaring the derived class
// as friend class in the base class
#include <iostream>
using namespace std;

// Forward declaration of the Derived class
class Derived;

// Base class
class Base {
private:
    int privateVar;

public:
    // Constructor to initialize privateVar
    Base(int val)
        : privateVar(val)
    {
    }
```

```
    // Declare Derived class as a friend
    friend class Derived;
};

// Derived class
class Derived {
public:
    // Function to display the private member of the base
    // class
    void displayPrivateVar(Base& obj)
    {
        // Accessing privateVar directly since Derived is a
        // friend of Base
        cout << "Value of privateVar in Base class: "
             << obj.privateVar << endl;
    }

    // Function to modify the private member of the base
    // class
    void modifyPrivateVar(Base& obj, int val)
    {
        // Modifying privateVar directly since Derived is a
        // friend of Base
        obj.privateVar = val;
    }
};

int main()
{
    // Create an object of Base class
    Base baseObj(10);

    // Create an object of Derived class
    Derived derivedObj;

    // Display the initial value of privateVar
    derivedObj.displayPrivateVar(baseObj);

    // Modify the value of privateVar
    derivedObj.modifyPrivateVar(baseObj, 20);

    // Display the modified value of privateVar
    derivedObj.displayPrivateVar(baseObj);

    return 0;
}
```

---

virtual base class

1. A virtual base class in C++ is used to solve the "diamond problem" that arises with multiple inheritance.

2. The diamond problem occurs when a derived class inherits from two classes that both inherit from the same base class.

3. When a class is specified as a virtual base class, it ensures that only one instance of the base class is shared among all the derived classes.

4. This way, the properties and behaviors of the base class are included only once in the final derived class, eliminating ambiguity.

5. For example, consider classes A, B, and C, where B and C both inherit from A, and D inherits from both B and C. If A is declared as a virtual base class, D will have only one instance of A, even though it inherits from both B and C.

Abstract classes

1. Abstract classes in C++ are classes that cannot be instantiated on their own because they contain one or more pure virtual functions.

2. Abstract classes are used to define a common interface for a group of related classes.

3. They typically include method declarations that define the behavior of a class, but leave the actual implementation to the derived classes.

4. Abstract classes are often used in conjunction with inheritance to create a hierarchy of classes where each derived class provides its own implementation of the abstract methods.

```
#include <iostream>

// Abstract base class Shape
class Shape {
public:
    // Pure virtual function to calculate area
```

```
    virtual double area() const = 0;

    // Virtual destructor (recommended for abstract classes)
    virtual ~Shape() {}
};

// Derived class Circle inheriting from Shape
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function area for Circle
    double area() const override {
        return 3.14159 * radius * radius; // Area of a circle: π * r^2
    }
};

// Derived class Rectangle inheriting from Shape
class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    // Implementation of the pure virtual function area for Rectangle
    double area() const override {
        return width * height; // Area of a rectangle: width * height
    }
};

int main() {
    // Example usage of polymorphism
    Circle c(5.0);
    Rectangle r(4.0, 3.0);

    // Pointer to Shape base class can point to any derived class
object
    Shape* shape1 = &c;
    Shape* shape2 = &r;

    // Calling area() through base class pointers
   cout << "Area of Circle: " << shape1->area() << endl;
```

```
   cout << "Area of Rectangle: " << shape2->area() << endl;

    return 0;
}
```

Nesting of classes

1. Nesting of classes in C++ refers to the practice of defining one class inside another class.

2. The nested class can access private members of the outer class, enhancing encapsulation.

3. Related classes can be grouped together, improving code organization and readability.

4. The nested class can be made private or protected, restricting access to it from outside the outer class.

5. Nesting classes can be particularly useful when you have a class that is closely related to another and is unlikely to be used outside of it.

6. It helps in maintaining clean and logical class relationships within your codebase.

```
#include <iostream>

// Outer class
class Outer {
private:
    int outer_private_data;

    // Nested class
    class Nested {
    private:
        int nested_private_data;

    public:
        void setNestedData(int data) {
            nested_private_data = data;
        }

        void display() {
            std::cout << "Nested private data: " <<
nested_private_data << std::endl;
```

```
        }
    };

public:
    Outer() : outer_private_data(0) {}

    void setOuterData(int data) {
        outer_private_data = data;
    }

    void display() {
        std::cout << "Outer private data: " << outer_private_data <<
std::endl;
    }

    // Accessing nested class functionality
    void nestedFunctionality() {
        Nested nestedObj;
        nestedObj.setNestedData(100);
        nestedObj.display();
    }
};

int main() {
    Outer outerObj;
    outerObj.setOuterData(50);
    outerObj.display();

    // Accessing nested class functionality through Outer class method
    outerObj.nestedFunctionality();

    return 0;
}
```

Explanation:

1. **Outer Class (`Outer`)**: This class contains a private member `outer_private_data` and a nested class `Nested`.
2. **Nested Class (`Nested`)**: This class is defined inside `Outer` and has its own private member `nested_private_data`. It provides methods `setNestedData()` and `display()` to set and display its private data.
3. **Usage in `Outer` Class**:
    - `nestedFunctionality()` method demonstrates how to create an instance of `Nested` class (`nestedObj`) within `Outer` class and access its methods.

Unit 5 (Polymorphism)

pointer

1. A pointer in C++ is a variable that holds the memory address of another variable.
2. Unlike regular variables that store actual data values, pointers store the addresses where those values are located in memory.
3. This allows for dynamic memory management and efficient array manipulation.
4. Pointers enable functions to modify variables outside their local scope by passing addresses instead of values, facilitating pass-by-reference.
5. They are crucial for creating complex data structures like linked lists, trees, and graphs.
6. Pointers can also point to other pointers, creating multi-level data structures.
7. However, improper use of pointers can lead to issues like memory leaks, segmentation faults, and undefined behavior, making them a powerful yet challenging feature of C++.
8. Understanding pointers is essential for mastering low-level programming and achieving optimal performance in C++.

---

virtual function

1. A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class.

2. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

3. Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.

4. They are mainly used to achieve Runtime polymorphism.

5. Functions are declared with a virtual keyword in a base class.

6. The resolving of a function call is done at runtime.

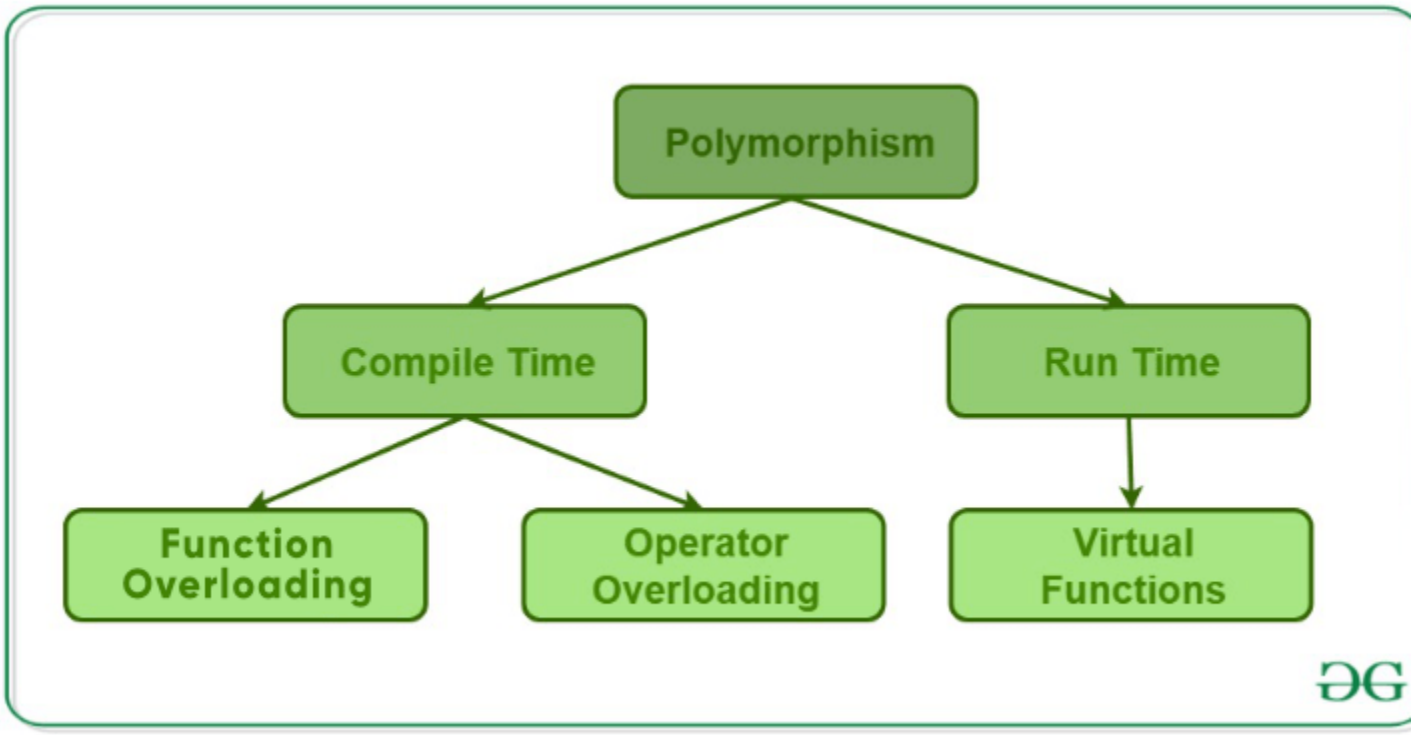
```
// C++ program to illustrate
// concept of Virtual Functions

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

```
class base {
public:
virtual void print() { cout << "print base class\n"; }

void show() { cout << "show base class\n"; }
};

class derived : public base {
public:
void print() { cout << "print derived class\n"; }

void show() { cout << "show derived class\n"; }
};

int main()
{
base* bptr;
derived d;
bptr = &d;

// Virtual function, binded at runtime
bptr->print();

// Non-virtual function, binded at compile time
bptr->show();

return 0;
}
```

Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

There are two types of polymorphism:



1. Compile-Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

A. Function Overloading

- When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading.
- Functions can be overloaded by changing the number of arguments or/and changing the type of arguments.
- In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name.

B. Operator Overloading

- C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
- For example, we can make use of the addition operator (+) for string class to concatenate two strings.
- We know that the task of this operator is to add two operands.
- So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

2. Runtime Polymorphism
This type of polymorphism is achieved by Function Overriding.

A. Function overriding

- Function overriding in C++ occurs when a derived class provides a specific implementation for a function that is already defined in its base class.
- This allows the derived class to replace or extend the behavior of the base class function.
- The function in the derived class must have the same name, return type, and parameters as the function in the base class.
- The base class function must be declared as virtual for the derived class function to override it. This ensures that the correct function is called based on the object's actual type at runtime.

---

pure virtual function

1. A pure virtual function in C++ is a virtual function that has no implementation in the base class and must be overridden in any derived class.

2. It is declared by assigning 0 to the function declaration in the base class.

3. Pure virtual functions make a class abstract, meaning that it cannot be instantiated on its own and is intended to be a blueprint for derived classes.

4. Pure virtual functions enforce a contract for the derived classes, ensuring that they implement specific behavior.

5. This is crucial for designing flexible and extensible software architectures, as it allows for polymorphism and the ability to define common interfaces for different classes.

Here's an example:

```
class Base {
public:
    virtual void show() = 0; // Pure virtual function
};
```

- In this case, `show` is a pure virtual function.
- Any class deriving from `Base` must provide an implementation for `show` to be instantiated.
- If a derived class does not override this function, it too becomes an abstract class.

Unit 6 (Working with Files, Console I/O Operations)

streams

In C++, streams are used to perform input and output operations. The Standard Library defines a hierarchy of stream classes, which are used for various types of I/O operations.
Basic Stream Classes

1. `istream`:
    - Used for input operations.
    - Derived classes: `ifstream` (input file stream), `istringstream` (input string stream).
2. `ostream`:
    - Used for output operations.
    - Derived classes: `ofstream` (output file stream), `ostringstream` (output string stream).
3. `iostream`:
    - Used for both input and output operations.
    - Derived classes: `fstream` (file stream), `stringstream` (string stream).

Unformatted I/O operations

Unformatted I/O operations in C++ are used for low-level input and output operations where data is read or written in a raw, unprocessed form. These operations are typically faster than formatted I/O operations because they bypass the format checking and conversion processes.

1. `istream::get`:

- Reads a single character or a block of characters.
- Overloaded versions allow reading into a single character, an array, or a stream buffer.

```
std::ifstream file("example.txt");
char ch;
file.get(ch);
std::cout << ch << std::endl; // Reads a single character
file.close();
```

2. `istream::read`:

- Reads a specified number of characters into a buffer.

```
std::ifstream file("example.txt", std::ios::binary);
char buffer[100];
file.read(buffer, 100); // Reads 100 characters
std::cout.write(buffer, file.gcount()); // Writes the number of
characters read
file.close();
```

Formatted Console I/O Operations

Formatted console I/O operations in C++ involve using various manipulators and functions to control the formatting of input and output. These operations are primarily handled using the `iostream` library, which provides extensive support for formatting.

1. Using the `<<` Operator:

- The `<<` operator is used for output operations, allowing you to chain multiple outputs.

```
int num = 42;
double pi = 3.14159;
std::cout << "Number: " << num << ", Pi: " << pi << std::endl;
```

2. Using Manipulators:

- `std::endl`: Inserts a newline and flushes the stream.
- `std::setw`: Sets the width of the next input/output field.
- `std::setprecision`: Sets the precision for floating-point numbers.

```
#include <iostream>
#include <iomanip>

int main() {
    double pi = 3.14159;
    std::cout << std::fixed << std::setprecision(2) << pi <<
std::endl; // 3.14
    std::cout << std::setw(10) << std::left << 42 << std::endl; //
"42        "
    std::cout << std::hex << 255 << std::endl; // ff
    return 0;
```

```
}
```

3. Using Member Functions:

- `.precision()`: Sets the precision for floating-point numbers.
- `.width()`: Sets the width of the next input/output field.

```
#include <iostream>

int main() {
    double pi = 3.14159;
    std::cout.precision(3);
    std::cout << pi << std::endl; // 3.14
    std::cout.width(10);
    std::cout.fill('*');
    std::cout << 42 << std::endl; // "********42"
    return 0;
}
```

File handling

1. File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory (Hard disk).

2. We give input to the executing program and the execution program gives back the output.

3. The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called stream.

4. In other words, streams are nothing but the flow of data in a sequence.

5. The input and output operation between the executing program and the devices like keyboard and monitor are known as “console I/O operation”.

6. The input and output operation between the executing program and files are known as “disk I/O operation”.

Classes for File stream

1. ios:-

- ios stands for input output stream.
- This class is the base class for other classes in this class hierarchy.
- This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

2. istream:-

- istream stands for input stream.
- This class is derived from the class ‘ios’.
- This class handle input stream.
- The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
- This class declares input functions such as get(), getline() and read().

3. ostream:-

- ostream stands for output stream.
- This class is derived from the class ‘ios’.
- This class handle output stream.
- The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
- This class declares output functions such as put() and write().

Reading and Closing files

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
    // Using ifstream to open a file for reading
    ifstream inputFile("input.txt");

    // Check if the file opened successfully
```

```
    if (!inputFile.is_open()) {
        cerr << "Error opening input file" << endl;
        return 1;
    }

    // Using ofstream to open a file for writing
    ofstream outputFile("output.txt");

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cerr << "Error opening output file" << endl;
        return 1;
    }

    // Using fstream to open a file for both reading and writing
    fstream ioFile("iofile.txt", ios::in | ios::out | ios::app);

    // Check if the file opened successfully
    if (!ioFile.is_open()) {
        cerr << "Error opening io file" << endl;
        return 1;
    }

    // Perform file operations...

    // Close the files
    inputFile.close();
    outputFile.close();
    ioFile.close();

    return 0;
}
```

Unit 7 (Exception Handling)

Exception handling

1. In C++, exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution.

2. The process of handling these exceptions is called exception handling.

3. Using the exception handling mechanism, the control from one part of the program where the exception occurred can be transferred to another part of the code.

4. So basically using exception handling in C++, we can handle the exceptions so that our program keeps running.

5. There are two types of exceptions in C++

- Synchronous: Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with, such as dividing a number by zero.
- Asynchronous: Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.

Need for exception handling

1. Separation of Error Handling Code: Exception handling separates error-handling code from regular code, making the logic easier to understand and maintain.

2. Handling Unexpected Errors: It provides a structured way to handle unexpected errors that might occur during runtime, such as file I/O errors, memory allocation failures, or network issues.

3. Improving Code Readability: By reducing the clutter of error-handling code within the main logic, exceptions improve code readability and focus on the primary functionality.

4. Facilitating Debugging: With exceptions, debugging is more straightforward since the call stack is unwound, and the point where the exception was thrown is clearly identified. This helps in quickly pinpointing the source of the errors.

5. Propagating Errors Up the Call Stack: Exceptions can propagate errors up the call stack, allowing a higher-level function to handle errors that occurred deep within the call hierarchy. This eliminates the need for every function to handle every possible error condition explicitly

---

try and catch

The try and catch keywords come in pairs: We use the try block to test some code and If the code throws an exception we will handle it in our catch block.

1. try in C++

The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.

2. catch in C++

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

3. throw in C++

An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

```
// C++ program to demonstate the use of try,catch and throw
// in exception handling.

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

int main()
{
```

```
// try block
try {
int numerator = 10;
int denominator = 0;
int res;

// check if denominator is 0 then throw runtime
// error.
if (denominator == 0) {
throw runtime_error(
"Division by zero not allowed!");
}

// calculate result if no exception occurs
res = numerator / denominator;
//[printing result after division
cout << "Result after division: " << res << endl;
}
// catch block to catch the thrown exception
catch (const exception& e) {
// print the exception
cout << "Exception " << e.what() << endl;
}

return 0;
}
```

Exception in Constructors and Destructors

1. Exceptions in Constructors:

Constructors can throw exceptions if they encounter errors during the initialization of an object. If a constructor throws an exception, the object is considered not to have been created, and its destructor will not be called. However, any previously constructed sub-objects (base class and member objects) will have their destructors called automatically.

```
#include <iostream>
#include <stdexcept>
```

```
using namespace std;

class Resource {
public:
    Resource() {
        cout << "Resource acquired\n";
    }
    ~Resource() {
        cout << "Resource released\n";
    }
};

class MyClass {
private:
    Resource res;
public:
    MyClass() {
        throw runtime_error("Constructor failed!");
    }
};

int main() {
    try {
        MyClass obj;
    } catch (const exception& e) {
        cerr << "Exception caught: " << e.what() << '\n';
    }
    return 0;
}
```

- In this example, if the `MyClass` constructor throws, the `Resource` destructor will be called to release the resource.

2. Exceptions in Destructors:

Destructors should not throw exceptions. If a destructor throws an exception during stack unwinding (when another exception is already propagating), the program will terminate (`std::terminate` will be called). This is because C++ does not allow multiple exceptions to propagate simultaneously.

```
#include <iostream>
#include <stdexcept>
```

```
using namespace std;

class MyClass {
public:
    ~MyClass() {
        try {
            // Code that might throw
            throw runtime_error("Destructor error!");
        } catch (const exception& e) {
            cerr << "Exception caught in destructor: " << e.what() <<
'\n';
            // Handle the exception, but do not let it propagate
        }
    }
};

int main() {
    try {
        MyClass obj;
        throw runtime_error("Main function error!");
    } catch (const exception& e) {
        cerr << "Exception caught: " << e.what() << '\n';
    }
    return 0;
}
```

- In this example, the destructor of `MyClass` catches any exceptions internally and prevents them from propagating, ensuring that the program does not terminate unexpectedly.

Unit 8 (Templates and Standard Template Library)

Understanding Templates

Imagine you want to write a function that returns the maximum of two numbers. You'd need to write separate functions for different data types (int, float, double, etc.):

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

double max(double a, double b) {
    return (a > b) ? a : b;
}

// And so on for each type...
```

This can be cumbersome and repetitive. Templates solve this problem by allowing you to write a single function that works with any data type.

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

- `template <typename T>`: This tells the compiler that `T` is a placeholder for a data type.

- `T max(T a, T b)`: This is the function definition, where `T` will be replaced with the actual data type when the function is called.

You can now use the `max` function with any data type:

```
int a = 5, b = 10;
cout << max(a, b) <<endl; // Works with int

double x = 5.5, y = 10.1;
cout << max(x, y) << endl; // Works with double
```

Templates

1. Templates in C++ allow for the creation of generic functions and classes, enabling code reuse and type safety.

2. They let you write a function or class to work with any data type without sacrificing performance.

3. A function template is defined with the template keyword followed by a parameter list in angle brackets.

4. Templates increase flexibility and reduce code duplication.

5. However, they can complicate debugging and increase compilation times due to code generation for each type used.

6. They are a cornerstone of C++'s powerful type system, enabling generic programming.

Class templates

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

// Class template
template <typename T>
class Box {
private:
    T value;

public:
    Box(T val) : value(val) {}

    void setValue(T val) {
        value = val;
    }

    T getValue() {
        return value;
    }

    void display() {
        cout << "Value: " << value << endl;
    }
};
```

```
int main() {
    // Creating a Box object for int
    Box<int> intBox(123);
    intBox.display();

    // Creating a Box object for double
    Box<double> doubleBox(456.78);
    doubleBox.display();

    // Creating a Box object for string
    Box<string> stringBox("Hello, world!");
    stringBox.display();

    return 0;
}
```

In this example:

- We define a template class `Box` that can hold a value of any data type.
- The class has a constructor, a setter (`setValue`), a getter (`getValue`), and a display function (`display`).
- In the `main` function, we create `Box` objects for different data types (`int`, `double`, and `string`) and display their values.

Function templates

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

// Function template
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // Using the function template with int
    int intResult = add(5, 3);
    cout << "intResult: " << intResult << endl;

    // Using the function template with double
    double doubleResult = add(2.5, 1.3);
```

```
    cout << "doubleResult: " << doubleResult << endl;

    // Using the function template with string
    string stringResult = add(string("Hello, "), string("world!"));
    cout << "stringResult: " << stringResult << endl;

    return 0;
}
```

In this example:

- We define a function template `add` that can take two arguments of any data type and return their sum.
- In the `main` function, we use the `add` function with different data types (`int`, `double`, and `string`), demonstrating how the same template function can be used with various types.

Member Functions

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

// Class with member function template
class Calculator {
public:
    // Member function template
    template <typename T>
    T add(T a, T b) {
        return a + b;
    }

    // Member function template
    template <typename T>
    T multiply(T a, T b) {
        return a * b;
    }
};

int main() {
    Calculator calc;

    // Using the member function template with int
```

```
    int intAddResult = calc.add(5, 3);
    int intMulResult = calc.multiply(5, 3);
    cout << "intAddResult: " << intAddResult << endl;
    cout << "intMulResult: " << intMulResult << endl;

    // Using the member function template with double
    double doubleAddResult = calc.add(2.5, 1.3);
    double doubleMulResult = calc.multiply(2.5, 1.3);
    cout << "doubleAddResult: " << doubleAddResult << endl;
    cout << "doubleMulResult: " << doubleMulResult << endl;

    // Using the member function template with string
    string stringAddResult = calc.add(string("Hello, "),
string("world!"));
    // Note: The multiply operation is not defined for strings, so we
won't use it here
    cout << "stringAddResult: " << stringAddResult << endl;

    return 0;
}
```

In this example:

- We define a `Calculator` class with two member function templates: `add` and `multiply`.
- Each member function template can operate on any data type.
- In the `main` function, we create a `Calculator` object and use its member function templates with different data types (`int`, `double`, and `string`).
- Note that while the `add` function works for all these types, the `multiply` function is not defined for `string` types, so we only use it with numeric types.

Standard Template Library

1. The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.

2. It is a library of container classes, algorithms, and iterators.

3. It is a generalized library and so, its components are parameterized. Working knowledge of template classes is a prerequisite for working with STL.

4. The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs.

5. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.

6. One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types.

Applications of Container Classes

1. `vector`

- **Dynamic Arrays**: `vector` is often used as a dynamic array where the size can be changed during runtime. It is efficient for random access and adding elements at the end.
- **Data Storage**: Used to store a collection of objects of the same type.

2. `list`

- **Doubly Linked List**: `list` is useful when frequent insertions and deletions are required, especially in the middle of the container.
- **Data Management**: Suitable for applications where data integrity during modifications is important, like undo functionality.

3. `deque`

- **Double-ended Queue**: `deque` supports fast insertions and deletions at both ends. It can be used for applications that require a queue with fast access from both ends.
- **Task Scheduling**: Useful in implementing task schedulers.

4. `set`

- **Unique Elements Storage**: `set` ensures all elements are unique and automatically sorted. It is ideal for storing a collection of unique items.
- **Fast Lookup**: Provides logarithmic time complexity for insertion, deletion, and lookup, making it useful for membership testing.

5. `map`

- **Associative Arrays**: `map` is used to store key-value pairs with unique keys. It is helpful for applications that require fast retrieval based on keys.
- **Dictionary Implementation**: Commonly used for implementing dictionaries where the key is the word and the value is the definition.