Question Bank			
Subject	**Object Oriented Programming using C++**	**Code**	**CIC-211**
Community	**SnapED Community**		

1. What is the difference between dynamic memory allocation and static memory allocation?

Sol:

Difference between Static and Dynamic Memory Allocation

S.No	Static Memory Allocation	Dynamic Memory Allocation
1	When the allocation of memory performs at the compile time, then it is known as static memory.	When the memory allocation is done at the execution or run time, then it is called dynamic memory allocation.
2	The memory is allocated at the compile time.	The memory is allocated at the runtime.
3	In static memory allocation, while executing a program, the memory cannot be changed.	In dynamic memory allocation, while executing a program, the memory can be changed.
4	Static memory allocation is preferred in an array.	Dynamic memory allocation is preferred in the linked list.
5	It saves running time as it is fast.	It is slower than static memory allocation.
6	Static memory allocation allots memory from the stack.	Dynamic memory allocation allots memory from the heap.
7	Once the memory is allotted, it will remain from the beginning to end of the program.	Here, the memory can be alloted at any time in the program.

8	Static memory allocation is less efficient as compared to Dynamic memory allocation.	Dynamic memory allocation is more efficient as compared to the Static memory allocation.
9	This memory allocation is simple.	This memory allocation is complicated.

2. Discuss the term function overloading. Write a program using function overloading on subtracting two given integer matrices.

Sol:

Function overloading in C++ allows you to define multiple functions with the same name but with different parameters. The compiler determines which function to call based on the number or type of arguments provided. Here's an example of function overloading for subtracting two integer matrices:

```
#include <iostream>

class Matrix {
public:
    // Function to subtract two integers
    int subtract(int a, int b) {
        return a - b;
    }

    // Function to subtract two matrices
    void subtract(int matrix1[3][3], int matrix2[3][3], int result[3][3]) {
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                result[i][j] = matrix1[i][j] - matrix2[i][j];
            }
        }
    }

    // Function to display a matrix
    void displayMatrix(int matrix[3][3]) {
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                std::cout << matrix[i][j] << " ";
            }
            std::cout << std::endl;
        }
    }
};

int main() {
    Matrix mat;

    // Subtract two integers
    int resultInt = mat.subtract(10, 5);
```

```
    std::cout << "Subtraction result (int): " << resultInt << std::endl;

    // Subtract two matrices
    int matrix1[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int matrix2[3][3] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};
    int resultMatrix[3][3];

    mat.subtract(matrix1, matrix2, resultMatrix);

    std::cout << "Matrix 1:" << std::endl;
    mat.displayMatrix(matrix1);

    std::cout << "Matrix 2:" << std::endl;
    mat.displayMatrix(matrix2);

    std::cout << "Subtraction result (matrix):" << std::endl;
    mat.displayMatrix(resultMatrix);

    return 0;
}
```

3. Differentiate between static binding and dynamic binding.

Sol:

Static Binding (Early Binding):

- **Resolution at Compile Time:** Also known as early binding, static binding involves the association of a function call with a specific function implementation at compile time.
- **Determined by Compiler:** The compiler determines which function to call based on the function name, signature, and the static type of the object or reference.
- **Example:** Non-virtual functions and function overloading use static binding.

```
class Base {
public:
    void display() {
        // Static binding
        std::cout << "Display from Base class" << std::endl;
    }
};

int main() {
    Base obj;
    obj.display();  // Static binding
    return 0;
}
```

Dynamic Binding (Late Binding):

- **Resolution at Runtime:** Also known as late binding, dynamic binding involves determining the function implementation to be called at runtime.
- **Involves Polymorphism:** Achieved through the use of virtual functions and pointers or references to base class objects.
- **Example:** Virtual functions enable dynamic binding.

```
class Base {
public:
   virtual void display() {
      // Dynamic binding
      std::cout << "Display from Base class" << std::endl;
   }
};

class Derived : public Base {
public:
   void display() override {
      // Dynamic binding
      std::cout << "Display from Derived class" << std::endl;
   }
};

int main() {
   Base* ptr = new Derived();
   ptr->display();  // Dynamic binding
   delete ptr;
   return 0;
}
```

4. How to resolve conflicts in naming between multiple parent classes if they are called from a child?

Sol: conflicts in naming between multiple parent classes can be resolved using the scope resolution operator (`::`). The scope resolution operator allows you to specify which class's member you are referring to when there is ambiguity. This is particularly important when a derived class inherits from multiple parent classes, and these classes have members with the same name.

```
#include <iostream>

class Parent1 {
public:
   void display() {
      std::cout << "Display from Parent1" << std::endl;
   }
};

class Parent2 {
public:
   void display() {
      std::cout << "Display from Parent2" << std::endl;
   }
};

class Child : public Parent1, public Parent2 {
public:
   void show() {
      // Resolve conflicts using the scope resolution operator
      Parent1::display();  // Call display from Parent1
```

```
        Parent2::display();  // Call display from Parent2
    }
};

int main() {
    Child obj;
    obj.show();
    return 0;
}
```

In this example, the `Child` class inherits from both `Parent1` and `Parent2`, and both parent classes have a `display` function. In the `show` function of the `Child` class, conflicts are resolved by using the scope resolution operator to explicitly specify which `display` function to call.

5. Differentiate between compile time polymorphism and runtime polymorphism. Give a suitable example.

Compile-Time Polymorphism (Static Binding):

Compile-time polymorphism, also known as static binding or early binding, occurs during the compilation phase of the program. It is achieved through function overloading and operator overloading. The decision about which function to call is made by the compiler based on the function signature or the operator used.

Example of Compile-Time Polymorphism:

```
#include <iostream>

class CompileTimePolymorphism {
public:
    // Function Overloading
    void display(int num) {
        std::cout << "Integer: " << num << std::endl;
    }

    void display(double num) {
        std::cout << "Double: " << num << std::endl;
```

```
    }
};

int main() {
    CompileTimePolymorphism obj;

    // Compiler decides which display function to call based on the argument type
    obj.display(5);       // Calls display(int)
    obj.display(3.14);    // Calls display(double)

    return 0;
}
```

In this example, the `CompileTimePolymorphism` class has two `display` functions with different parameter types. The compiler determines at compile time which version of the function to call based on the argument type.

Run-Time Polymorphism (Dynamic Binding):

Run-time polymorphism, also known as dynamic binding or late binding, occurs during the runtime of the program. It is achieved through virtual functions and is associated with inheritance and polymorphism. The decision about which function to call is made at runtime based on the actual type of the object.

Example of Run-Time Polymorphism:

```
#include <iostream>

class Shape {
public:
    // Virtual function
    virtual void draw() {
        std::cout << "Drawing a shape" << std::endl;
    }
};
```

```
class Circle : public Shape {
public:
    // Overriding the virtual function
    void draw() override {
        std::cout << "Drawing a circle" << std::endl;
    }
};

class Square : public Shape {
public:
    // Overriding the virtual function
    void draw() override {
        std::cout << "Drawing a square" << std::endl;
    }
};

int main() {
    Shape* shape;

    Circle circle;
    Square square;

    // Polymorphic behavior at runtime
    shape = &circle;
    shape->draw();  // Calls draw() of Circle

    shape = &square;
    shape->draw();  // Calls draw() of Square

    return 0;
}
```

6. What is exception handling? How are exceptions handled in C++? Illustrate with an example. What are the differences between synchronous and asynchronous exceptions?

Exception handling is a mechanism in C++ that allows you to deal with unexpected or error conditions during the execution of a program in a controlled and organized manner. Exceptions provide a way to transfer control from one part of the program to another in the event of an error.

Handling Exceptions in C++:

1. **`try` Block:** A `try` block encloses the code that might throw an exception.
2. **`catch` Block:** A `catch` block follows the `try` block and handles the exceptions thrown within it.
3. **`throw` Statement:** The `throw` statement is used to throw an exception when a specific error condition occurs.

4. **`std::exception` Class:** Many standard exceptions derive from the **`std::exception`** class, allowing you to catch them polymorphically.

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

double divide(double numerator, double denominator) {
    if (denominator == 0) {
        throw std::runtime_error("Division by zero is not allowed");
    }
    return numerator / denominator;
}

int main() {
    try {
        double result = divide(10.0, 2.0);
        std::cout << "Result: " << result << std::endl;

        // This will throw an exception
        result = divide(8.0, 0.0);
        std::cout << "This line won't be executed." << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Exception caught: " << e.what() << std::endl;
    }

    return 0;
}
```

In this example, the **`divide`** function checks for division by zero and throws a **`std::runtime_error`** if encountered. The **`try`** block in the **`main`** function catches this exception and prints an error message.

Synchronous vs. Asynchronous Exceptions:

1. **Synchronous Exceptions:**
   - **Definition:** Synchronous exceptions occur as a direct result of the execution of an instruction.
   - **Example:** Division by zero, accessing an invalid memory location.
   - **Handling:** Typically handled using try-catch blocks.
2. **Asynchronous Exceptions:**
   - **Definition:** Asynchronous exceptions occur outside the normal flow of control, often triggered by external events.
   - **Example:** Signals, hardware faults, termination requests.
   - **Handling:** More challenging to handle, often require special mechanisms (like signal handlers on Unix systems).

7. Differentiate between text and binary file? How are they handled differently? Give an example of five functions to read and write into both types of files.

Text File vs. Binary File:

Text File:

- **Content:** Contains human-readable text.
- **Representation:** Stores characters using character encoding (e.g., ASCII or UTF-8).
- **Viewable:** Can be opened and viewed with a text editor.
- **Example:** `.txt` files.

Binary File:

- **Content:** Contains non-human-readable data, including binary data (e.g., images, executables).
- **Representation:** Stores raw binary data without character encoding.
- **Viewable:** Not easily viewable with a text editor.
- **Example:** `.jpg`, `.exe`, `.dat` files.

Handling Text and Binary Files in C++:

Five Functions for Handling Text Files:

1. `std::ofstream` **(Output File Stream):**
   - Used for writing data to a text file.

```
#include <fstream>
std::ofstream outFile("example.txt");
outFile << "Hello, World!";
outFile.close();
```

2. `std::ifstream` **(Input File Stream):**
   - Used for reading data from a text file.

```
#include <fstream>
std::ifstream inFile("example.txt");
std::string content;
inFile >> content;
inFile.close();pp
```

3. `std::fstream` **(File Stream):**
   - Used for both reading and writing data to/from a text file.

```
#include <fstream>
std::fstream file("example.txt", std::ios::in | std::ios::out);
std::string content;
file >> content;
file << "Appending text.";
file.close();
```

4. `std::getline()` **Function:**
   - Reads a line from a text file.

```
#include <fstream>
```

```
std::ifstream inFile("example.txt");
std::string line;
std::getline(inFile, line);
inFile.close();
```

6. `std::endl` **Manipulator:**
   - Writes a newline character and flushes the output buffer.

Five Functions for Handling Binary Files:

1. `std::ofstream` **(Output File Stream):**
   - Used for writing binary data to a file.

```
cppCopy code
 include <fstream> std::ofstream outFile "data.bin" s ios::binary ; int a = 42
      write reinterpret_cast char >( ), sizeof int ) ou close ;
```

2. `std::ifstream` **(Input File Stream):**
   - Used for reading binary data from a file.

```
cppCopy code
 include <fstream> inFile "data.bin" int read reinterpret_cast char
sizeof int close
```

3. `std::fstream` **(File Stream):**
   - Used for both reading and writing binary data to/from a file.

```
#include <fstream>
std::fstream file("data.bin", std::ios::in | std::ios::out | std::ios::binary);
int data;
file.read(reinterpret_cast<char*>(&data), sizeof(int));
file.write(reinterpret_cast<char*>(&data), sizeof(int));
file.close();
```

4. `seekg()` **and** `seekp()` **Functions:**
   - Moves the file pointer to a specified position.

```
#include <fstream>
std::fstream file("data.bin", std::ios::in | std::ios::out | std::ios::binary);
file.seekg(0, std::ios::beg); // Move to the beginning
```

5. `read()` **and** `write()` **Functions:**
   - Read/write a specified number of bytes from/to a binary file.

```
#include <fstream>
std::fstream file("data.bin", std::ios::in | std::ios::out | std::ios::binary);
char buffer[100];
file.read(buffer, sizeof(buffer));
file.write(buffer, sizeof(buffer));
file.close();
```

8. What is generic programming? What are its advantages?

Generic programming is a programming paradigm that emphasizes creating algorithms and data structures that can work with different data types, rather than being tied to a specific

data type. The idea is to write code in a way that is independent of the data types it operates on, allowing it to be reused with various types without modification.

Advantages of Generic Programming:

1. **Reusability:**
   - Generic programming promotes the creation of reusable code. Algorithms and data structures can be written once and used with different data types without modification.
2. **Flexibility:**
   - Code written in a generic manner is more flexible and adaptable to different scenarios. It can be applied to a wide range of data types without requiring changes to the code.
3. **Maintainability:**
   - Generic code is often more maintainable because changes to the code can be localized. When a generic algorithm needs to be modified, the change typically only needs to be made in one place, affecting all instances of its use.
4. **Code Size Reduction:**
   - Generic programming can lead to a reduction in code size. Instead of duplicating similar code for different data types, a single generic implementation can handle multiple types.
5. **Improved Readability:**
   - Generic code tends to be more readable and expressive, as it focuses on the logic of the algorithm rather than the details of specific data types.
6. **Algorithm Independence:**
   - Generic programming allows developers to separate algorithms from data types. This means that improvements or changes to algorithms don't necessitate modifications to the underlying data types.
7. **Enhanced Abstraction:**
   - Generic programming encourages the use of high-level abstractions, making it easier to reason about and understand the functionality of the code.
8. **Consistency Across Data Types:**
   - When using generic programming, the same algorithm behaves consistently across different data types, which can contribute to more predictable and reliable software.
9. **Support for Multiple Paradigms:**
   - Generic programming complements other programming paradigms, such as object-oriented programming (OOP) or procedural programming. It can be used in conjunction with these paradigms to provide a well-rounded and flexible approach to software development.
10. **Standardization:**
    - Many modern programming languages provide support for generic programming through features like templates in C++, generics in Java, and generics in C#. This standardization promotes consistent and widely applicable coding practices.

9. Explain dynamic memory allocation in C++.

Dynamic memory allocation in C++ is the process of allocating and deallocating memory during program execution explicitly. Unlike static memory allocation, where memory is allocated at compile-time, dynamic memory allocation allows you to allocate memory at runtime, providing more flexibility and efficiency in managing memory resources.

The key operators used for dynamic memory allocation in C++ are `new` and `delete`. These operators allow you to allocate memory on the heap, and they should be used with caution to avoid memory leaks.

10. Difference between overloaded functions and overridden functions.

Function Overloading	***Function Overriding***
Function Overloading provides multiple definitions of the function by changing signature.	Function Overriding is the redefinition of base class function in its derived class with same signature.
An example of compile time polymorphism.	An example of run time polymorphism.
Function signatures should be different.	Function signatures should be the same.
Overloaded functions are in same scope.	Overridden functions are in different scopes.
Overloading is used when the same function has to behave differently depending upon parameters passed to them.	Overriding is needed when derived class function has to do some different job than the base class function.
A function has the ability to load multiple times.	A function can be overridden only a single time.
In function overloading, we don't need inheritance.	In function overriding, we need an inheritance concept.

11. What is ambiguity resolution in class inheritance? When do you encounter such a situation and how is it handled? Explain with an example.

Ambiguity resolution in class inheritance occurs when a derived class inherits from two or more base classes, and there is a naming conflict in the derived class. This conflict arises when a member (function or variable) is declared in multiple base classes, and the derived class doesn't provide an explicit override or clarification on how to use these conflicting members.

Here's an example to illustrate ambiguity resolution in class inheritance:

```
#include <iostream>

// Base class A
class A {
public:
    void display() {
        std::cout << "Class A" << std::endl;
    }
};

// Base class B
class B {
public:
    void display() {
        std::cout << "Class B" << std::endl;
    }
};

// Derived class from A and B
class Derived : public A, public B {
public:
    // Uncommenting the line below will resolve the ambiguity
    // using A::display;

    // Uncommenting the line below will resolve the ambiguity
    // using B::display;

    // This function does not resolve the ambiguity
    // void display() {
    //     std::cout << "Derived Class" << std::endl;
    // }
};

int main() {
    Derived derivedObj;

    // Uncommenting the line below will resolve the ambiguity
    // derivedObj.A::display();

    // Uncommenting the line below will resolve the ambiguity
    // derivedObj.B::display();
```

```
    // This line will result in ambiguity if the resolution is not done
    derivedObj.display();

    return 0;
}
```

In this example:

- There are two base classes, **`A`** and **`B`**, both having a **`display`** member function.
- The **`Derived`** class inherits from both **`A`** and **`B`**.
- The **`display`** function in the **`Derived`** class does not provide an explicit override for the conflicting **`display`** functions from **`A`** and **`B`**.
- In the **`main`** function, if you uncomment the lines using **`A::display()`** or **`B::display()`**, you are explicitly resolving the ambiguity by specifying which version of the function to use.

If you don't resolve the ambiguity, attempting to call **`display`** on a **`Derived`** object directly will result in a compilation error. The compiler doesn't know which version of **`display`** to use. By explicitly using **`A::display()`** or **`B::display()`**, you provide clarification on which base class's version of the function to call.

12. What do you mean by a template member function? Write a program to define the function template for calculating the square of given numbers with different data types.

A template member function is a member function of a class that is defined as a template. Just like function templates, which are not tied to any class, template member functions provide a way to write a generic function that can work with different data types. They are useful when you want to apply the same algorithm or operation to various types within a class.

Here's an example of a C++ program with a template member function to calculate the square of a given number:

```
#include <iostream>



// Class with a template member function

template <typename T>

class Calculator {

public:
```

```
    // Template member function to calculate the square
    T calculateSquare(T num) {
        return num * num;
    }
};

int main() {
    // Creating instances of the Calculator class with different data types
    Calculator<int> intCalculator;
    Calculator<float> floatCalculator;
    Calculator<double> doubleCalculator;

    // Calculating squares for different data types
    int intResult = intCalculator.calculateSquare(5);
    float floatResult = floatCalculator.calculateSquare(3.5f);
    double doubleResult = doubleCalculator.calculateSquare(2.5);

    // Displaying results
    std::cout << "Square of 5 (int): " << intResult << std::endl;
    std::cout << "Square of 3.5 (float): " << floatResult << std::endl;
    std::cout << "Square of 2.5 (double): " << doubleResult << std::endl;

    return 0;
```

```
}
```

13. Consider an example of declaring the examination result. Design three classes: student, exam and result. The student class has data members such as those representing roll numbers, names etc. Create the class exam by inheriting the student class. The exam class adds data members representing the marks scored in six subjects. Derive the Result from Exam class and it has its own data members such as total marks. Write an interactive program to model this relationship.

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

// Student class
class Student {
protected:
    int rollNumber;
    std::string name;

public:
    // Parameterized constructor
    Student(int roll, const std::string& n) : rollNumber(roll), name(n) {}

    // Display student information
    void displayStudent() const {
        std::cout << "Roll Number: " << rollNumber << "\nName: " << name << std::endl;
    }
};

// Exam class inheriting from Student
class Exam : public Student {
protected:
    int marks[6]; // Marks in six subjects

public:
    // Parameterized constructor
    Exam(int roll, const std::string& n, const int m[]) : Student(roll, n) {
        // Copy marks to the member variable
        for (int i = 0; i < 6; ++i) {
            marks[i] = m[i];
        }
    }

    // Display student information along with marks
    void displayExam() const {
        displayStudent(); // Call base class display function
        std::cout << "Marks in Subjects: ";
        for (int i = 0; i < 6; ++i) {
            std::cout << marks[i] << " ";
        }
        std::cout << std::endl;
```

```
    }
};

// Result class inheriting from Exam
class Result : public Exam {
private:
    int totalMarks;

public:
    // Parameterized constructor
    Result(int roll, const std::string& n, const int m[]) : Exam(roll, n, m) {
        // Calculate total marks
        totalMarks = 0;
        for (int i = 0; i < 6; ++i) {
            totalMarks += marks[i];
        }
    }

    // Display result along with total marks
    void displayResult() const {
        displayExam(); // Call base class display function
        std::cout << "Total Marks: " << totalMarks << std::endl;
    }
};

int main() {
    // Get student information and marks interactively
    int rollNumber;
    std::string name;
    int marks[6];

    std::cout << "Enter Roll Number: ";
    std::cin >> rollNumber;
    std::cout << "Enter Name: ";
    std::cin.ignore(); // Ignore newline from previous input
    std::getline(std::cin, name);

    std::cout << "Enter Marks in Six Subjects:\n";
    for (int i = 0; i < 6; ++i) {
        std::cout << "Subject " << i + 1 << ": ";
        std::cin >> marks[i];
    }

    // Create Result object and display the result
    Result studentResult(rollNumber, name, marks);
    std::cout << "\nDisplaying Student Result:\n";
    studentResult.displayResult();

    return 0;
}
```

14. What are pure virtual functions? How do they differ from normal virtual functions?

Normal Virtual Functions:

- Normal virtual functions are defined in the base class and can be overridden by derived classes.
- They have a body in the base class, providing a default implementation.
- Derived classes may choose to override or not override the virtual function.
- Objects of the base class can be instantiated, but they cannot be instantiated if the class contains any pure virtual functions.

Pure Virtual Functions:

- Pure virtual functions are declared in the base class without providing an implementation (no body).
- Classes containing pure virtual functions are called abstract classes, and they cannot be instantiated.
- Derived classes must override pure virtual functions to become concrete (instantiable) classes.
- Pure virtual functions create a "contract" that derived classes must fulfill.

15. Write a C++ program to read the class object of student info such as name, age, gender, height and weight from the keyboard and to store them on a specified file using read() and write() functions. Again the same file is opened for reading and displaying the contents of the file on the screen.

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

class Student {
public:
    std::string name;
    int age;
    char gender;
    double height;
    double weight;

    // Function to read student information from keyboard
    void read() {
        std::cout << "Enter Name: ";
        std::getline(std::cin, name);

        std::cout << "Enter Age: ";
        std::cin >> age;

        std::cout << "Enter Gender (M/F): ";
        std::cin >> gender;
```

```
        std::cout << "Enter Height (in meters): ";
        std::cin >> height;

        std::cout << "Enter Weight (in kg): ";
        std::cin >> weight;
    }

    // Function to write student information to file
    void write(std::ofstream& outputFile) {
        outputFile << name << " " << age << " " << gender << " "
                   << height << " " << weight << "\n";
    }

    // Function to display student information
    void display() const {
        std::cout << "Name: " << name << "\nAge: " << age
                  << "\nGender: " << gender << "\nHeight: " << height
                  << "\nWeight: " << weight << std::endl;
    }
};

int main() {
    // Writing student information to file
    std::ofstream outputFile("student_info.txt");
    if (outputFile.is_open()) {
        Student student;
        student.read();
        student.write(outputFile);
        outputFile.close();
    } else {
        std::cerr << "Unable to open file for writing." << std::endl;
        return 1;
    }

    // Reading and displaying student information from file
    std::ifstream inputFile("student_info.txt");
    if (inputFile.is_open()) {
        Student studentFromFile;
        inputFile >> studentFromFile.name >> studentFromFile.age >> studentFromFile.gender
                  >> studentFromFile.height >> studentFromFile.weight;

        std::cout << "\nStudent Information Read from File:\n";
        studentFromFile.display();

        inputFile.close();
    } else {
        std::cerr << "Unable to open file for reading." << std::endl;
        return 1;
    }
```

```
    return 0;
}
```

16. Write short notes on the following:-

    a) Exception handling in C++
    b) Generic Classes
    c) Vectors

a) Exception Handling in C++:

a. Exception handling in C++ is a mechanism to handle runtime errors and abnormal conditions.
b. It involves three keywords: `try`, `catch`, and `throw`.
c. `try`: Contains the code that might throw an exception.
d. `catch`: Catches and handles the exception thrown in the `try` block.
e. `throw`: Explicitly throws an exception.

b) Generic Classes:

f. Generic classes, in the context of C++, are classes that are designed to work with different data types.
g. Templates in C++ facilitate the creation of generic classes and functions.
h. Example: `template <typename T> class Container { /* ... */ };`

c) Vectors:

i. Vectors in C++ are dynamic arrays that can grow or shrink in size.
j. Defined in the `<vector>` header.
k. Provide dynamic memory allocation, easy element access, and various utility functions.

17. Give the syntax of the following stream functions: getline() and write().

`getline()` Function:

```
std::getline(std::cin, variable);
```

Reads a line of input from the stream `std::cin` and stores it in the variable.

`write()` Function:

```
output_stream.write(buffer, size);
```

18. Compare inline functions and macros. Discuss which one should be used in which circumstances?

Inline Functions:

1. Defined using the `inline` keyword.
2. Follows the function call mechanism.
3. Type-safe and supports debugging.
4. Recommended for small functions.

Macros:

5. Defined using the `#define` preprocessor directive.
6. Direct text substitution during preprocessing.
7. Error-prone and challenging for debugging.

8. Recommended for simple, short, and performance-critical code.

19. What is super, base, parent, inherited class? Compare multilevel and multiple inheritances in C++.

Super Class:

1. Not a standard term in C++.
2. Sometimes used informally to refer to the base class in the context of inheritance.

Base Class:

3. The class from which another class (derived class) inherits.
4. Provides common attributes and behaviors.

Parent Class:

5. Synonymous with the base class.
6. The class that is inherited by another class.

Inherited Class:

7. Synonymous with the derived class.
8. The class that inherits from another class.

20. Explain inheritance and its applications. Explain the features of inheritance and polymorphism.

Inheritance:

1. Mechanism in C++ where a class inherits properties and behaviors from another class.
2. Types: Single, Multiple, Multilevel, Hierarchical, Hybrid.

Applications:

3. Code Reusability: Inherited class can reuse features of the base class.
4. Polymorphism: Enables dynamic method invocation through virtual functions.
5. Structuring Code: Provides a way to structure and organize code.

Features of Inheritance and Polymorphism:

6. **Inheritance:**
    1. **Code Reusability:** Inherited class can reuse features of the base class.
    2. **Access Control:** Public, protected, and private inheritance control access to base class members.
    3. **Derivation Types:** Single, Multiple, Multilevel, Hierarchical, Hybrid.
7. **Polymorphism:**
    1. **Virtual Functions:** Enable dynamic method invocation based on the actual object type.
    2. **Dynamic Binding:** Resolving function calls at runtime.
    3. **Abstract Classes:** Classes with pure virtual functions, promoting polymorphic behavior.

21. Write a C++ program to illustrate the array of pointers to objects.

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

```
class Student {
public:
    std::string name;
    int age;

    Student(const std::string& n, int a) : name(n), age(a) {}
};

int main() {
    const int arraySize = 3;
    Student* studentArray[arraySize];

    studentArray[0] = new Student("Alice", 20);
    studentArray[1] = new Student("Bob", 22);
    studentArray[2] = new Student("Charlie", 21);

    for (int i = 0; i < arraySize; ++i) {
        std::cout << "Student " << i + 1 << ": " << studentArray[i]->name << ", Age: " <<
studentArray[i]->age << std::endl;
    }

    // Cleanup: Release memory allocated for objects
    for (int i = 0; i < arraySize; ++i) {
        delete studentArray[i];
    }

    return 0;
}
```

22. Write a C++ program to illustrate function template and class template? Write a function template to find the minimum of three numbers.

```
#include <iostream>

// Function Template
template <typename T>
T findMinimum(T a, T b, T c) {
    return (a < b) ? ((a < c) ? a : c) : ((b < c) ? b : c);
}

// Class Template
template <typename T>
class Pair {
public:
    T first;
    T second;

    Pair(T f, T s) : first(f), second(s) {}
};

int main() {
```

```
    // Function Template Example
    int minInt = findMinimum(5, 3, 8);
    double minDouble = findMinimum(3.14, 2.71, 4.2);

    std::cout << "Minimum Integer: " << minInt << std::endl;
    std::cout << "Minimum Double: " << minDouble << std::endl;

    // Class Template Example
    Pair<int> intPair(10, 20);
    Pair<double> doublePair(3.5, 7.2);

    std::cout << "Integer Pair: " << intPair.first << ", " << intPair.second << std::endl;
    std::cout << "Double Pair: " << doublePair.first << ", " << doublePair.second << std::endl;

    return 0;
}
```

23. Explain aggregation and composition used in object oriented languages. Give their applications.

- **Aggregation:**
  - **Definition:** Aggregation represents a "has-a" relationship between objects, where one object contains another as a part.
  - **Example:** A university "has-a" collection of students. Students can exist independently of the university.
  - **Application:** Modeling relationships where one object is part of another but can exist independently.
- **Composition:**
  - **Definition:** Composition represents a stronger form of aggregation, where one object is composed of other objects, and the lifetime of the contained objects is controlled by the container.
  - **Example:** A car "has-a" engine. The engine is an integral part of the car, and it is created and destroyed with the car.
  - **Application:** Modeling relationships where the existence of one object is dependent on another.

24. Write a C++ program to illustrate operator overloading for binary and unary operators

```
#include <iostream>

class Complex {
public:
    double real;
    double imag;

    Complex(double r, double i) : real(r), imag(i) {}

    // Binary Operator Overloading
    Complex operator+(const Complex& other) const {
```

```
        return Complex(real + other.real, imag + other.imag);
    }

    // Unary Operator Overloading
    Complex operator-() const {
        return Complex(-real, -imag);
    }
};

int main() {
    Complex a(3.0, 4.0);
    Complex b(1.5, 2.5);

    Complex sum = a + b;
    Complex negA = -a;

    std::cout << "Sum: " << sum.real << " + " << sum.imag << "i" << std::endl;
    std::cout << "Negation of A: " << negA.real << " + " << negA.imag << "i" << std::endl;

    return 0;
}
```

.

25. Compare overloading and overriding of member functions. Explain with an example.

Overloading:

- **Definition:** Overloading occurs when two or more functions in the same scope have the same name but different parameters.
- **Example:**

```
int add(int a, int b);
double add(double a, double b);
```

Overriding:

- **Definition:** Overriding occurs when a derived class provides a specific implementation for a function that is already declared in its base class.
- **Example:**

```
class Base {
public:
    virtual void print() const;
};

class Derived : public Base {
public:
    void print() const override;
};
```

26. Write a C++ program to illustrate virtual functions. Mention the rules for virtual functions in C++.

```
#include <iostream>

class Base {
```

```
public:
    // Virtual function
    virtual void show() const {
        std::cout << "Base class show()" << std::endl;
    }
};

class Derived : public Base {
public:
    // Overriding the virtual function
    void show() const override {
        std::cout << "Derived class show()" << std::endl;
    }
};

int main() {
    Base* basePtr = new Derived();

    // Calls the show() function of the derived class (polymorphism)
    basePtr->show();

    delete basePtr;
    return 0;
}
```

27. Explain the persistent objects and multiple inheritances in C++.

Persistent Objects:

- **Definition:** Persistent objects refer to objects whose state persists beyond the duration of a program's execution. Typically, this involves storing object data in non-volatile storage, such as databases or files. Persistence allows data to be preserved between program runs.
- **Application:** Storing user preferences, saving and loading game state, maintaining user profiles in applications, etc.
- **Example:**

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

class Person {
public:
    std::string name;
    int age;

    // Serialization method to save object state to a file
    void saveToFile(const std::string& filename) const {
        std::ofstream outFile(filename);
        outFile << name << std::endl;
        outFile << age << std::endl;
        outFile.close();
    }
```

```
    // Deserialization method to load object state from a file
    void loadFromFile(const std: :string& filename) {
        std::ifstream inFile(filename);
        inFile >> name;
        inFile >> age;
        inFile.close();
    }
};

int main() {
    Person user;
    user.name = "John Doe";
    user.age = 25;

    // Save user data to a file
    user.saveToFile("user_data.txt");

    // Load user data from the file
    Person loadedUser;
    loadedUser.loadFromFile("user_data.txt");

    std::cout << "Loaded User: " << loadedUser.name << ", Age: " << loadedUser.age << std::endl;

    return 0;
}
```

28. Explain manipulators. Write a C++ program to illustrate user defined manipulators.

Manipulators in C++:

- **Explanation:** Manipulators in C++ are functions or objects that modify the behavior of the output stream or perform specific operations on the stream. They are used with the insertion (<<) and extraction (>>) operators to control the format of input and output.
- **Example:**

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

int main() {
    double pi = 3.141592653589793;

    // Using setw manipulator to set the field width
    std::cout << std::setw(10) << "PI Value: " << pi << std::endl;

    // Using setprecision manipulator to set the precision of floating-point output
    std::cout << "PI Value (precision 4): " << std::setprecision(4) << pi << std::endl;

    // Using fixed and setprecision to display fixed-point notation with precision
    std::cout << "PI Value (fixed, precision 2): " << std::fixed << std::setprecision(2) << pi << std::endl;
```

```
    return 0;
}
```

29. Explain the role of iterators and allocators used in STL. Describe the various components of STL in detail.

- **Role of Iterators:**
  - **Definition:** Iterators are objects that allow the traversal of elements in a container. They provide a uniform way to access the elements of various containers without exposing the underlying container implementation.
  - **Usage:** Used in algorithms to operate on container elements, providing a generic interface for algorithms to work with different containers.
  - **Example:** Iterating through elements of a vector or a list.
- **Role of Allocators:**
  - **Definition:** Allocators are responsible for memory allocation and deallocation for the elements stored in a container. They allow customization of memory management strategies for containers.
  - **Usage:** Provides flexibility to choose specific memory allocation policies or integrate with custom memory models.
  - **Example:** Allocating memory for elements in a vector.
- **Components of STL:**
  1. **Containers:** Data structures that store elements in a specific arrangement (e.g., vectors, lists).
  2. **Algorithms:** Functions that operate on ranges of elements defined by iterators (e.g., sort, find).
  3. **Iterators:** Objects that allow the traversal of elements in a container (e.g., input, output, random access).
  4. **Functors:** Objects that can be called as if they were functions (e.g., function objects, lambda expressions).
  5. **Allocators:** Objects responsible for memory allocation and deallocation for containers.
  6. **Adapters:** Classes that provide a different interface to existing classes (e.g., stack, queue).
  7. **Utilities:** Miscellaneous helper classes and functions (e.g., pair, tuple, swap).
  8. **Algorithm Complexity Requirements:** Describes the complexity requirements for each algorithm.

These components collectively form the Standard Template Library (STL) in C++, providing a powerful and generic framework for handling various data structures and algorithms.

30. Explain exception handling. Mention the process of handling uncaught exceptions.

Exception handling is a mechanism in C++ that allows you to deal with runtime errors in a more controlled and structured manner. Exceptions are abnormal conditions or errors that can occur during the execution of a program. The C++ language provides a set of keywords and constructs for handling exceptions:

1. **`try` Block:**
   - The `try` block is used to enclose a section of code where exceptions might occur.
   - If an exception occurs within the `try` block, the control is transferred to the appropriate `catch` block.
2. **`catch` Block:**

- The `catch` block is used to handle exceptions.
- It specifies the type of exception it can catch.
- If an exception of the specified type is thrown within the corresponding `try` block, the control transfers to the associated `catch` block.

3. **`throw` Statement:**
   - The `throw` statement is used to explicitly throw an exception.
   - It can be used in functions to indicate that an error has occurred.
4. **`throw` with No Argument:**
   - If `throw` is used with no argument, it rethrows the currently handled exception.

Example:

```
#include <iostream>

void divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw std::runtime_error("Division by zero!");
    }
    std::cout << "Result: " << numerator / denominator << std::endl;
}

int main() {
    try {
        divide(10, 2);
        divide(8, 0); // This will throw an exception
        divide(12, 3); // This won't be executed if an exception occurs before
    } catch (const std::exception& e) {
        std::cerr << "Exception caught: " << e.what() << std::endl;
    }

    // Continue with the rest of the program...

    return 0;
}
```

31. Write a C++ program to illustrate formatted I/O and unformatted 1/0 using stream functions.

In C++, formatted I/O involves using stream functions to format the input and output, while unformatted I/O deals with raw data. Below is a simple C++ program that illustrates both formatted and unformatted I/O using stream functions:

```
#include <iostream>
#include <iomanip> // For formatted I/O manipulators
#include <fstream> // For file I/O

int main() {
    // Formatted Output
    std::cout << "Formatted Output:" << std::endl;
    int intValue = 42;
    double doubleValue = 3.14159;
```

```
// Using setw for width and setprecision for floating-point precision
std::cout << std::setw(10) << "Integer:" << std::setw(5) << intValue << std::endl;
std::cout << std::setw(10) << "Double:" << std::setprecision(3) << doubleValue << std::endl;

// Unformatted Output
std::cout << "\nUnformatted Output:" << std::endl;
char str[] = "Hello, World!";

// Using write() for unformatted output
std::cout.write(str, sizeof(str) - 1); // sizeof(str) includes the null terminator, so we subtract 1

// Formatted Input
std::cout << "\n\nFormatted Input:" << std::endl;
int inputInt;
double inputDouble;

// Using extraction operator for formatted input
std::cout << "Enter an integer: ";
std::cin >> inputInt;

std::cout << "Enter a double: ";
std::cin >> inputDouble;

// Displaying the input values
std::cout << "You entered: " << inputInt << " and " << inputDouble << std::endl;

// Unformatted Input from File
std::cout << "\nUnformatted Input from File:" << std::endl;

// Writing data to a binary file
std::ofstream outFile("data.dat", std::ios::binary);
outFile.write(reinterpret_cast<char*>(&intValue), sizeof(int));
outFile.write(reinterpret_cast<char*>(&doubleValue), sizeof(double));
outFile.close();

// Reading data from the binary file
std::ifstream inFile("data.dat", std::ios::binary);
if (inFile.is_open()) {
    int readInt;
    double readDouble;

    inFile.read(reinterpret_cast<char*>(&readInt), sizeof(int));
    inFile.read(reinterpret_cast<char*>(&readDouble), sizeof(double));

    std::cout << "Read from file: " << readInt << " and " << readDouble << std::endl;

    inFile.close();
} else {
    std::cerr << "Error opening file for reading." << std::endl;
```

```
    }

    return 0;
}
```

In this program:

- Formatted Output: The program uses `setw` and `setprecision` manipulators for formatting integers and doubles.
- Unformatted Output: The program uses `write()` for unformatted output, writing a character array to the console.
- Formatted Input: The program uses the extraction operator (`>>`) for formatted input, taking an integer and a double from the user.
- Unformatted Input from File: The program writes an integer and a double to a binary file using `write()` and then reads the values back from the file using `read()`. The `reinterpret_cast` is used to convert the address of the variables to a `char*` for writing and reading raw bytes.

32. Explain the features of the following STL:
    a) Sequence containers
    b) Associative containers
    c) Derived containers

STL (Standard Template Library) Features:

a) **Sequence Containers:**

- **Definition:** Sequence containers are a type of container that organizes elements in a linear or sequential order. They maintain the order in which elements are inserted.
- **Examples:** `vector`, `list`, `deque`, `array`.
- **Features:**
    - **Ordered:** Elements are stored in a specific order.
    - **Random Access:** Efficient element access through iterators or index (for vectors and arrays).
    - **Dynamic Size:** Containers can grow or shrink dynamically.
    - **Efficient Insertion and Deletion:** Some sequence containers provide efficient insertion and deletion at both ends (e.g., `list`, `deque`).

b) **Associative Containers:**

- **Definition:** Associative containers are a type of container that organize elements using key-based associations. Each element has a unique key, and the elements are arranged based on this key.
- **Examples:** `set`, `map`, `multiset`, `multimap`.

- **Features:**
    - **Key-Based Organization:** Elements are organized based on a key, allowing for efficient retrieval.
    - **Unique Keys:** In `set` and `map`, keys are unique.
    - **Sorted Order:** Elements are often stored in sorted order based on keys.
    - **Fast Search:** Fast search, insertion, and deletion based on the key.
    - **Associative Relationships:** Establishing relationships between elements based on keys.

c) **Derived Containers:**

- **Definition:** Derived containers are not explicitly part of the standard terminology. It's possible that this term is being used in the context of containers that are derived or specialized versions of the basic STL containers.
- **Examples:** Containers that are derived or specialized from the basic STL containers.
- **Features:**
    - **Specialized Functionality:** Derived containers may provide additional or specialized functionality beyond the basic containers.
    - **Inheritance:** They often inherit properties and behaviors from the basic containers.
    - **Template Specialization:** Derived containers may involve template specialization to customize behavior.

33. What do you mean by dynamic binding? How is it useful in C++?

Dynamic Binding in C++: Dynamic binding, also known as late binding or runtime polymorphism, refers to the mechanism by which the selection of the actual function or method to be executed is determined at runtime. This is particularly associated with the use of virtual functions and pointers or references to base class objects. Dynamic binding allows for more flexibility in the handling of objects and their behavior, especially in the context of inheritance and polymorphism.

Usefulness of Dynamic Binding in C++: Dynamic binding is useful in C++ for several reasons:

1. **Polymorphism:** Dynamic binding is a key feature of polymorphism. It allows a program to work with objects of different classes through a common interface (base class) and ensures that the appropriate function is called based on the actual type of the object at runtime.
2. **Extensibility:** Dynamic binding facilitates the extension of a program by allowing the addition of new derived classes without modifying existing code. New classes can inherit from existing base classes and override their virtual functions to provide new implementations.
3. **Flexibility:** It allows for the creation of generic code that operates on base class pointers or references, making it possible to work with objects of various derived classes without knowing their exact types at compile time.

4. **Run-Time Decisions:** Dynamic binding enables decisions about which function to call to be deferred until runtime. This is in contrast to static binding (compile-time polymorphism), where the decision is made at compile time based on the static type of the object.

Example of Dynamic Binding:

```
#include <iostream>

class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a shape" << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle" << std::endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a square" << std::endl;
    }
};

int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Square();

    shape1->draw();  // Calls draw() of Circle at runtime
    shape2->draw();  // Calls draw() of Square at runtime

    delete shape1;
    delete shape2;

    return 0;
}
```

In this example, the `draw` function is declared as virtual in the base class `Shape`. When objects of derived classes (`Circle` and `Square`) are accessed through a base class pointer (`Shape*`), the actual function called is determined at runtime based on the type of the object being pointed to. This demonstrates dynamic binding in action.

34. Explain how base class member function can be invoked in a derived class if the derived class also has a member function with the same.

when a derived class has a member function with the same name as a member function in the base

class, the member function in the derived class overrides (hides) the member function in the base class. However, if you still want to invoke the base class's member function from within the derived class, you can use the scope resolution operator `::`.

```
#include <iostream>

// Base class
class Base {
public:
    void display() {
        std::cout << "Display from Base class" << std::endl;
    }
};

// Derived class
class Derived : public Base {
public:
    // Overriding the display function in the derived class
    void display() {
        std::cout << "Display from Derived class" << std::endl;
    }

    // Function to invoke the base class's display function
    void invokeBaseDisplay() {
        // Use the scope resolution operator to invoke the base class's display function
        Base::display();
    }
};

int main() {
    Derived derivedObj;

    // Calling the overridden display function in the derived class
    derivedObj.display(); // Output: Display from Derived class

    // Calling the base class's display function from within the derived class
    derivedObj.invokeBaseDisplay(); // Output: Display from Base class

    return 0;
}
```

- The **`Base`** class has a member function called **`display`**.
- The **`Derived`** class inherits publicly from **`Base`** and overrides the **`display`** function.
- The **`invokeBaseDisplay`** function in the **`Derived`** class uses the scope resolution operator (**`Base::display()`**) to explicitly call the **`display`** function from the base class.

35. What is an inheritance? What are the different visibility modes supported by C++. Explain with an example.

Inheritance in C++: Inheritance is a fundamental concept in object-oriented programming

(OOP) that allows a class (called the derived or child class) to inherit properties and behaviors from another class (called the base or parent class). The derived class can then extend or modify these inherited properties and behaviors. Inheritance promotes code reusability and supports the creation of a hierarchy of classes.

Visibility Modes in C++: C++ supports three visibility modes for the members of a base class when inherited by a derived class. These modes are:

1. **Public Inheritance:**
   - Public members of the base class remain public in the derived class.
   - Protected members of the base class remain protected in the derived class.
   - Private members of the base class are not accessible in the derived class.
2. **Protected Inheritance:**
   - Public and protected members of the base class become protected in the derived class.
   - Private members of the base class are not accessible in the derived class.
3. **Private Inheritance:**
   - Public and protected members of the base class become private in the derived class.
   - Private members of the base class are not accessible in the derived class.

Example: Let's consider an example with a `Person` class as the base class and a `Student` class as the derived class. The `Student` class inherits publicly, protectedly, and privately from the `Person` class, demonstrating the three visibility modes.

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

// Base class
class Person {
public:
    std::string name;
    int age;

    Person(const std::string& n, int a) : name(n), age(a) {}

    void display() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

// Derived class using public inheritance
class PublicInheritanceStudent : public Person {
public:
    int studentID;

    PublicInheritanceStudent(const std::string& n, int a, int id)
        : Person(n, a), studentID(id) {}
```

```
    void displayStudent() const {
        display(); // Accessing public member of base class
        std::cout << "Student ID: " << studentID << std::endl;
    }
};

// Derived class using protected inheritance
class ProtectedInheritanceStudent : protected Person {
public:
    int studentID;

    ProtectedInheritanceStudent(const std::string& n, int a, int id)
        : Person(n, a), studentID(id) {}

    void displayStudent() const {
        // display(); // Error: Accessing private member of base class
        std::cout << "Student ID: " << studentID << std::endl;
    }
};

// Derived class using private inheritance
class PrivateInheritanceStudent : private Person {
public:
    int studentID;

    PrivateInheritanceStudent(const std::string& n, int a, int id)
        : Person(n, a), studentID(id) {}

    void displayStudent() const {
        // display(); // Error: Accessing private member of base class
        std::cout << "Student ID: " << studentID << std::endl;
    }
};

int main() {
    // Public Inheritance
    PublicInheritanceStudent publicStudent("John", 20, 123);
    publicStudent.displayStudent();

    // Protected Inheritance
    // ProtectedInheritanceStudent protectedStudent("Jane", 22, 456);
    // protectedStudent.displayStudent(); // Error: display() is protected in the derived class

    // Private Inheritance
    // PrivateInheritanceStudent privateStudent("Alice", 25, 789);
    // privateStudent.displayStudent(); // Error: display() is private in the derived class

    return 0;
}
```

In this example:

- **`PublicInheritanceStudent`** inherits publicly from **`Person`**, allowing public and protected members to remain accessible in the derived class.
- **`ProtectedInheritanceStudent`** inherits protectedly, making public members protected and accessible within the derived class.
- **`PrivateInheritanceStudent`** inherits privately, making both public and protected members private and accessible only within the derived class.

36. What is a template function in C++? Write a program for template function overloading.

template function is a function that is parameterized with one or more types. This allows you to write functions that can work with different data types without having to write separate functions for each type. Template functions are defined using the **`template`** keyword, followed by the template parameter list.

Here's an example of a template function and template function overloading in C++:

```
#include <iostream>

// Template function to find the minimum of two values
template <typename T>
T findMin(T a, T b) {
    return (a < b) ? a : b;
}

// Template function overloading to find the minimum of three values
template <typename T>
T findMin(T a, T b, T c) {
    return findMin(findMin(a, b), c);
}

int main() {
    // Example of using template function with integers
    int intResult = findMin(5, 3);
    std::cout << "Minimum of 5 and 3: " << intResult << std::endl;

    // Example of using template function with doubles
    double doubleResult = findMin(2.5, 3.7);
    std::cout << "Minimum of 2.5 and 3.7: " << doubleResult << std::endl;

    // Example of using template function overloading with three integers
    int intResultThree = findMin(8, 3, 5);
    std::cout << "Minimum of 8, 3, and 5: " << intResultThree << std::endl;

    // Example of using template function overloading with three doubles
```

```
    double doubleResultThree = findMin(4.5, 6.2, 3.8);
    std::cout << "Minimum of 4.5, 6.2, and 3.8: " << doubleResultThree << std::endl;

    return 0;
}
```

In this example:

- The `findMin` template function is defined to find the minimum of two values of the same type.
- The `findMin` template function is overloaded to find the minimum of three values of the same type by calling the two-value version recursively.
- The `main` function demonstrates the usage of these template functions with different data types (integers and doubles).

37. Write a program that reads content from one file and copies that content in another file.

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

int main() {
    std::string inputFileName, outputFileName;

    // Get input and output file names from the user
    std::cout << "Enter the name of the input file: ";
    std::cin >> inputFileName;

    std::cout << "Enter the name of the output file: ";
    std::cin >> outputFileName;

    // Input file stream to read content
    std::ifstream inputFile(inputFileName, std::ios::in);

    if (!inputFile.is_open()) {
        std::cerr << "Error opening input file: " << inputFileName << std::endl;
        return 1; // Return with an error code
    }

    // Output file stream to write content
    std::ofstream outputFile(outputFileName, std::ios::out);

    if (!outputFile.is_open()) {
        std::cerr << "Error opening output file: " << outputFileName << std::endl;
        inputFile.close(); // Close the input file
        return 1; // Return with an error code
    }
```

```
    // Read content from the input file and copy it to the output file
    char ch;
    while (inputFile.get(ch)) {
        outputFile.put(ch);
    }

    // Close the file streams
    inputFile.close();
    outputFile.close();

    std::cout << "Content copied successfully from " << inputFileName << " to " << outputFileName
<< std::endl;

    return 0; // Return with success
}
```

1. The user is prompted to enter the names of the input and output files.
2. An input file stream (`ifstream`) is opened to read content from the input file.
3. An output file stream (`ofstream`) is opened to write content to the output file.
4. Characters are read from the input file one by one and written to the output file until the end of the file is reached.
5. The file streams are closed after the operation is completed.

38. Write a program to read and write values through objects using file handling, List various file models available in C++. How to open and close the file? Explain using example code.

sol

file handling is done through the use of file streams. There are three file models available in C++:

1. **Text File Model:** Files that store data in human-readable text format.
2. **Binary File Model:** Files that store data in a binary format.
3. **Random Access File Model:** Allows reading and writing at any position within the file.

Here's an example program demonstrating how to read and write values through objects using file handling in C++. This example focuses on text file handling:

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

class Student {
public:
    std::string name;
    int age;
    double marks;

    // Parameterized constructor
    Student(const std::string& n, int a, double m) : name(n), age(a), marks(m) {}
```

```
    // Default constructor
    Student() : age(0), marks(0.0) {}

    // Function to display student details
    void display() const {
        std::cout << "Name: " << name << ", Age: " << age << ", Marks: " << marks << std::endl;
    }
};

int main() {
    // Writing objects to a text file
    std::ofstream outFile("students.txt", std::ios::out);

    if (!outFile.is_open()) {
        std::cerr << "Error opening file for writing." << std::endl;
        return 1;
    }

    Student student1("Alice", 20, 85.5);
    Student student2("Bob", 22, 78.0);

    outFile << student1.name << " " << student1.age << " " << student1.marks << std::endl;
    outFile << student2.name << " " << student2.age << " " << student2.marks << std::endl;

    outFile.close();

    // Reading objects from the text file
    std::ifstream inFile("students.txt", std::ios::in);

    if (!inFile.is_open()) {
        std::cerr << "Error opening file for reading." << std::endl;
        return 1;
    }

    Student readStudent1, readStudent2;

    inFile >> readStudent1.name >> readStudent1.age >> readStudent1.marks;
    inFile >> readStudent2.name >> readStudent2.age >> readStudent2.marks;

    inFile.close();

    // Displaying read student details
    std::cout << "Details of Read Students:" << std::endl;
    readStudent1.display();
    readStudent2.display();

    return 0;
}
```

In this example:

1. The **`Student`** class represents a student with name, age, and marks.
2. Objects of the **`Student`** class (**`student1`** and **`student2`**) are written to a text file (**`students.txt`**) using an output file stream (**`ofstream`**).
3. Objects are then read back from the file using an input file stream (**`ifstream`**).
4. The read student details are displayed.

39. Write a program for the following: Define a class bank account with current and saving bank accounts as inherited classes. Class bank accounts should have following data members: Account number, name, Balance Amount and Member functions: to initialize the value, to deposit and withdraw amount after checking the minimum balance.

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

class BankAccount {
protected:
    int accountNumber;
    std::string accountHolderName;
    double balance;

public:
    BankAccount(int accNumber, const std::string& accHolderName, double initialBalance)
        : accountNumber(accNumber), accountHolderName(accHolderName), balance(initialBalance)
{}

    // Function to deposit amount
    void deposit(double amount) {
        balance += amount;
        std::cout << "Amount deposited. New balance: " << balance << std::endl;
    }

    // Function to withdraw amount with a check on minimum balance
    virtual void withdraw(double amount) {
        const double minBalance = 100.0; // Assuming a minimum balance requirement

        if (balance - amount >= minBalance) {
            balance -= amount;
            std::cout << "Amount withdrawn. New balance: " << balance << std::endl;
        } else {
            std::cout << "Insufficient funds. Withdrawal not allowed." << std::endl;
        }
    }

    // Function to display account details
    void display() const {
        std::cout << "Account Number: " << accountNumber << std::endl;
        std::cout << "Account Holder Name: " << accountHolderName << std::endl;
        std::cout << "Balance: " << balance << std::endl;
```

```
    }
};

class CurrentAccount : public BankAccount {
public:
    CurrentAccount(int accNumber, const std::string& accHolderName, double initialBalance)
        : BankAccount(accNumber, accHolderName, initialBalance) {}

    // Overriding withdraw function for CurrentAccount
    void withdraw(double amount) override {
        // Allow withdrawal without minimum balance check for CurrentAccount
        balance -= amount;
        std::cout << "Amount withdrawn from Current Account. New balance: " << balance << std::endl;
    }
};

class SavingsAccount : public BankAccount {
public:
    SavingsAccount(int accNumber, const std::string& accHolderName, double initialBalance)
        : BankAccount(accNumber, accHolderName, initialBalance) {}

    // Overriding withdraw function for SavingsAccount
    void withdraw(double amount) override {
        // Implement any specific rules for withdrawal from SavingsAccount (if needed)
        BankAccount::withdraw(amount); // Call the base class withdraw function
    }
};

int main() {
    // Example usage
    CurrentAccount currentAcc(12345, "John Doe", 1000.0);
    SavingsAccount savingsAcc(54321, "Jane Smith", 500.0);

    currentAcc.display();
    currentAcc.deposit(500.0);
    currentAcc.withdraw(300.0);

    std::cout << "\n";

    savingsAcc.display();
    savingsAcc.deposit(200.0);
    savingsAcc.withdraw(700.0); // This will not be allowed due to the minimum balance check

    return 0;
}
```