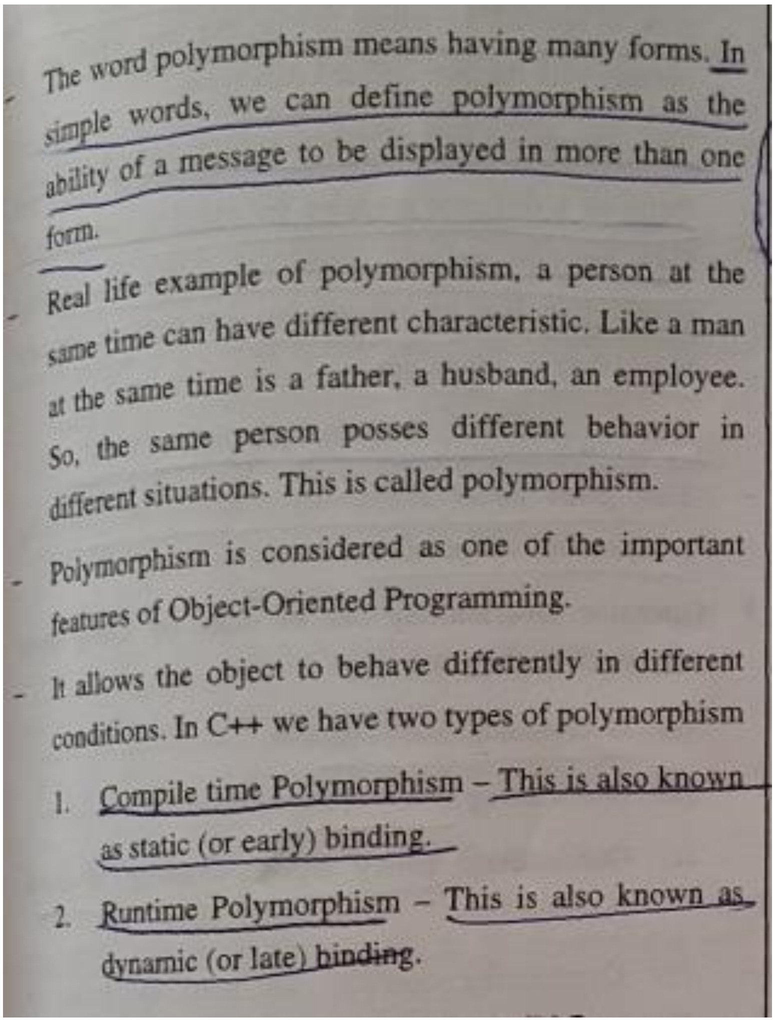
UNIT 3

POLYMORPHISM

1) Explain the polymorphism feature of OOP. What are the different ways to achieve polymorphism in C++ Language? Explain them along with examples.



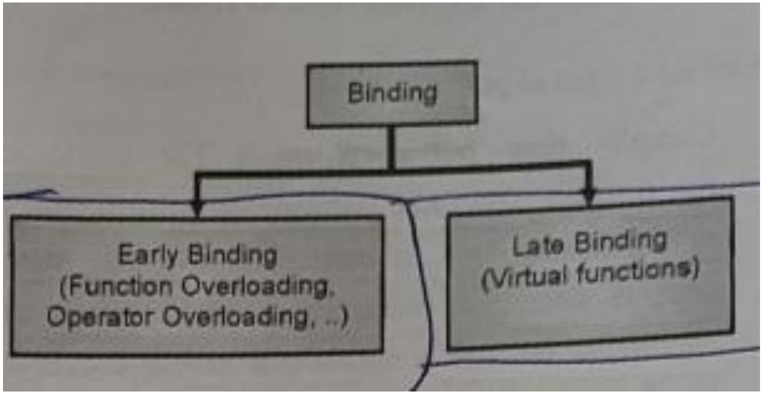
- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

- Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So, the same person posses different behavior in different situations. This is called polymorphism.

- Polymorphism is considered as one of the important features of Object-Oriented Programming.

- It allows the object to behave differently in different conditions. In C++ we have two types of polymorphism

1. Compile time Polymorphism – This is also known as static (or early) binding.
2. Runtime Polymorphism – This is also known as dynamic (or late) binding.

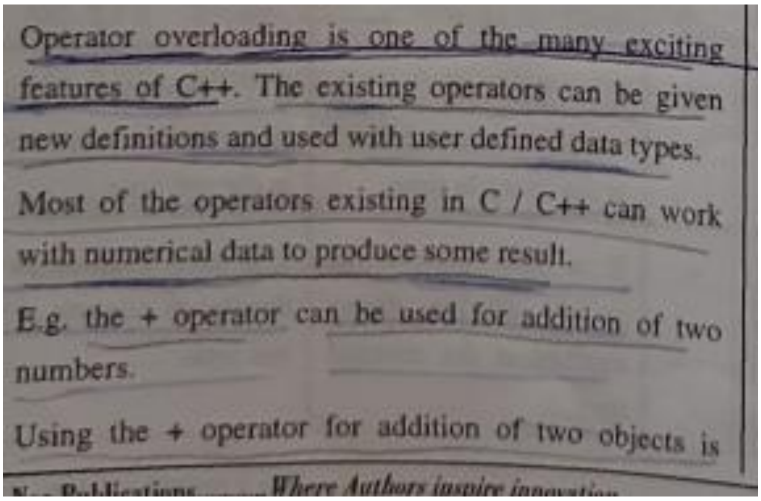

Binding
Early Binding (Function Overloading, Operator Overloading, ..)
Late Binding (Virtual functions)

2) **What is operator overloading? Write a program to overload '+' operator for adding two complex numbers which are object of below complex class.**

Class Complex {

Private: int real, imag;

};

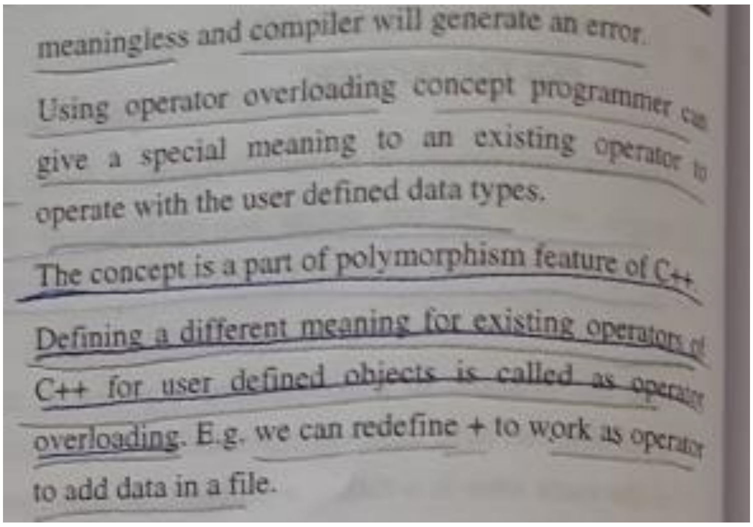


Operator overloading is one of the many exciting features of C++. The existing operators can be given new definitions and used with user defined data types.

Most of the operators existing in C / C++ can work with numerical data to produce some result.

E.g. the + operator can be used for addition of two numbers.

Using the + operator for addition of two objects is

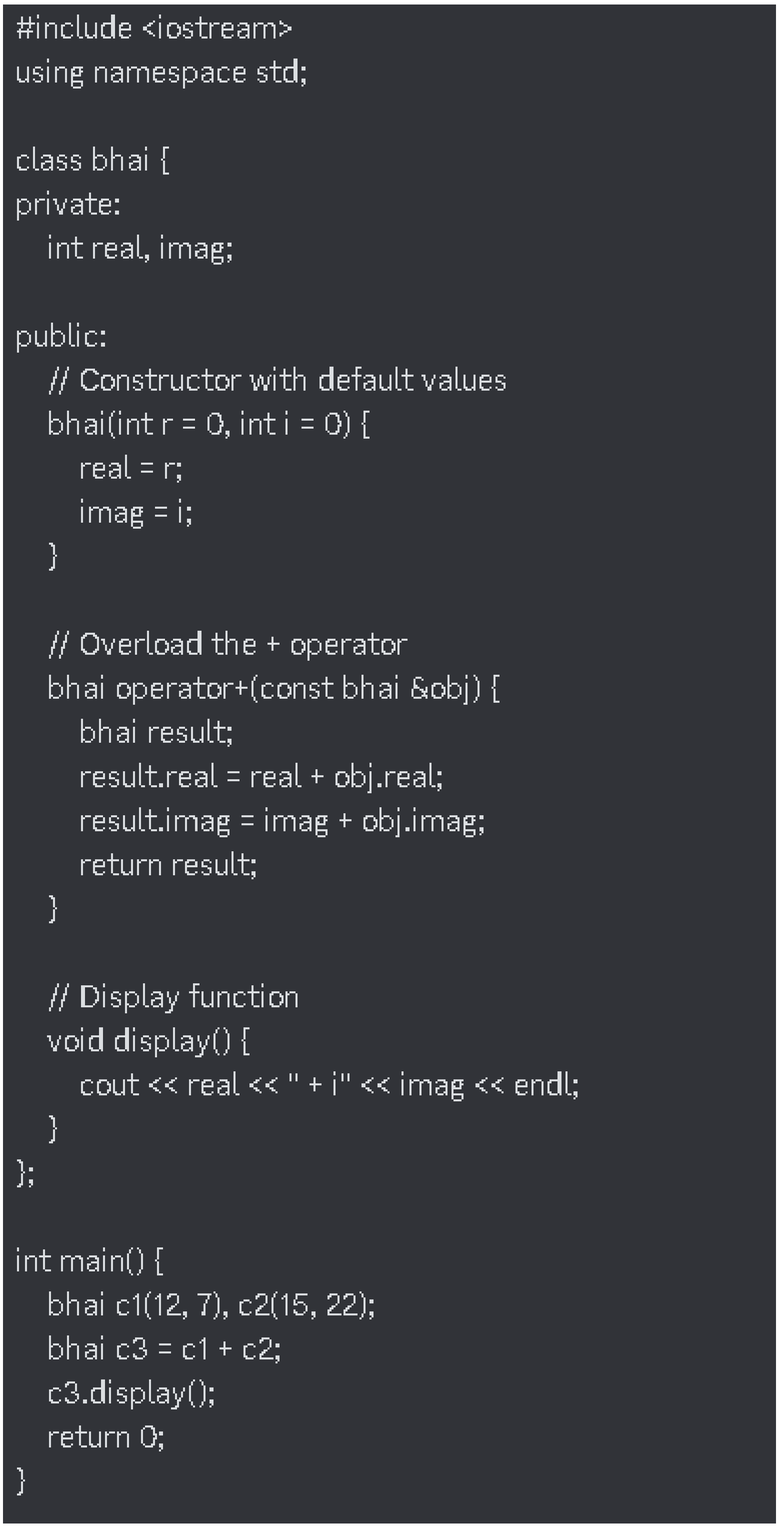


meaningless and compiler will generate an error.

Using operator overloading concept programmer can give a special meaning to an existing operator to operate with the user defined data types.

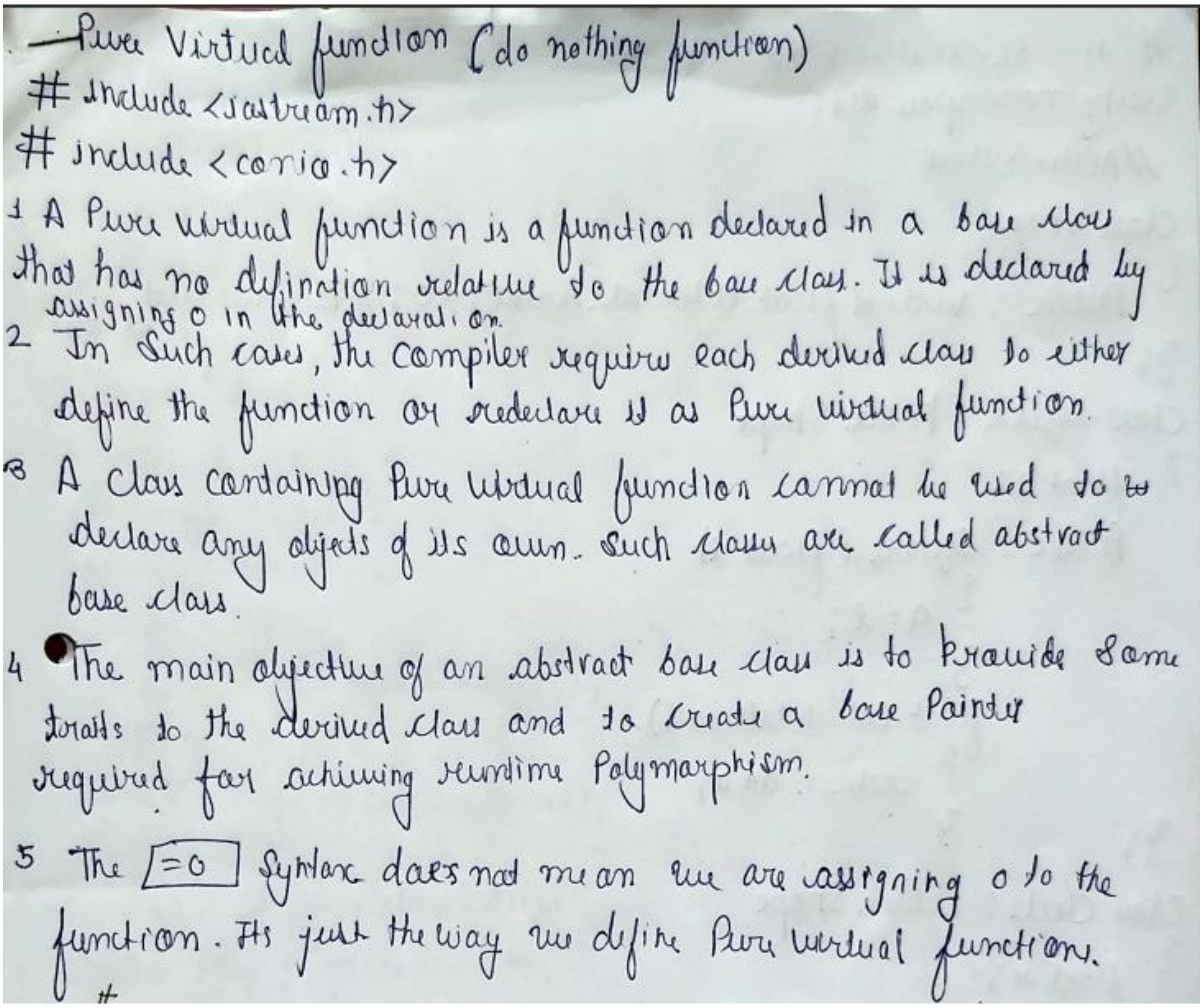
The concept is a part of polymorphism feature of C++.

Defining a different meaning for existing operators of C++ for user defined objects is called as operator overloading. E.g. we can redefine + to work as operator to add data in a file.



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

class bhai {
private:
    int real, imag;

public:
    // Constructor with default values
    bhai(int r = 0, int i = 0) {
        real = r;
        imag = i;
    }

    // Overload the + operator
    bhai operator+(const bhai &obj) {
        bhai result;
        result.real = real + obj.real;
        result.imag = imag + obj.imag;
        return result;
    }

    // Display function
    void display() {
        cout << real << " + i" << imag << endl;
    }
};

int main() {
    bhai c1(12, 7), c2(15, 22);
    bhai c3 = c1 + c2;
    c3.display();
    return 0;
}
```

3) What is Pure virtual function? Illustrate the use of Pure virtual function



—Pure Virtual function (do nothing function)

# include <iostream.h>

# include <conio.h>

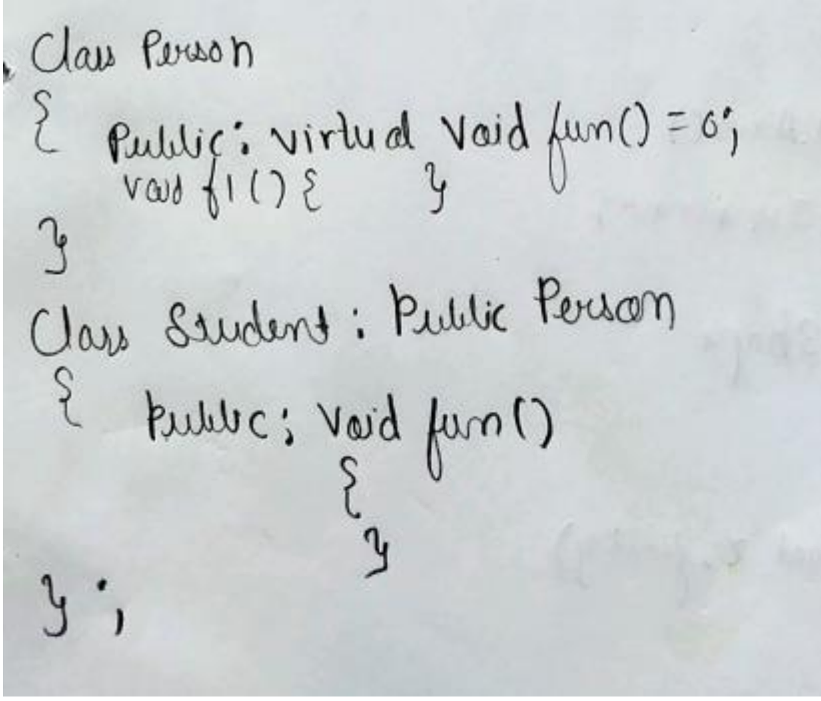
1 A Pure virtual function is a function declared in a base class that has no difination relative to the base class. It is declared by assigning 0 in the declaration.

2 In such cases, the compiler require each derived class to either define the function or redeclare it as Pure virtual function.

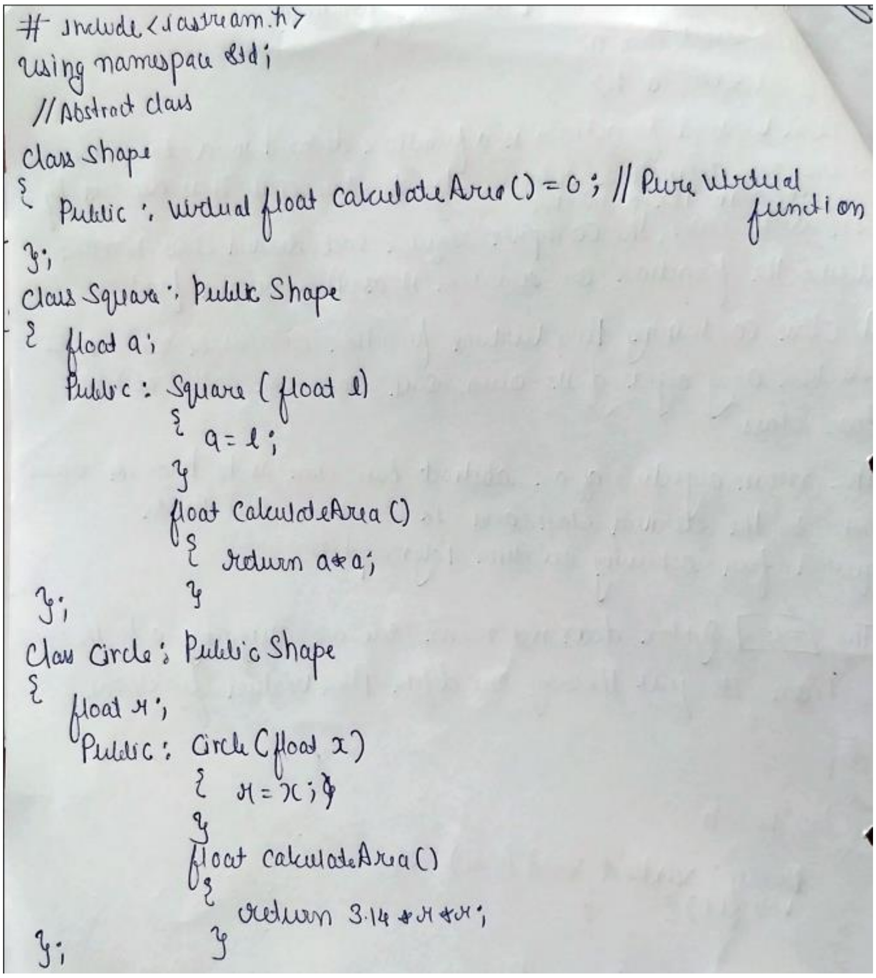
3 A class containing Pure virtual function cannot be used to to declare any objects of its own. Such classes are called abstract base class.

4 The main objective of an abstract base class is to provide some traits to the derived class and to create a base Pointer required for achieving runtime Polymorphism.

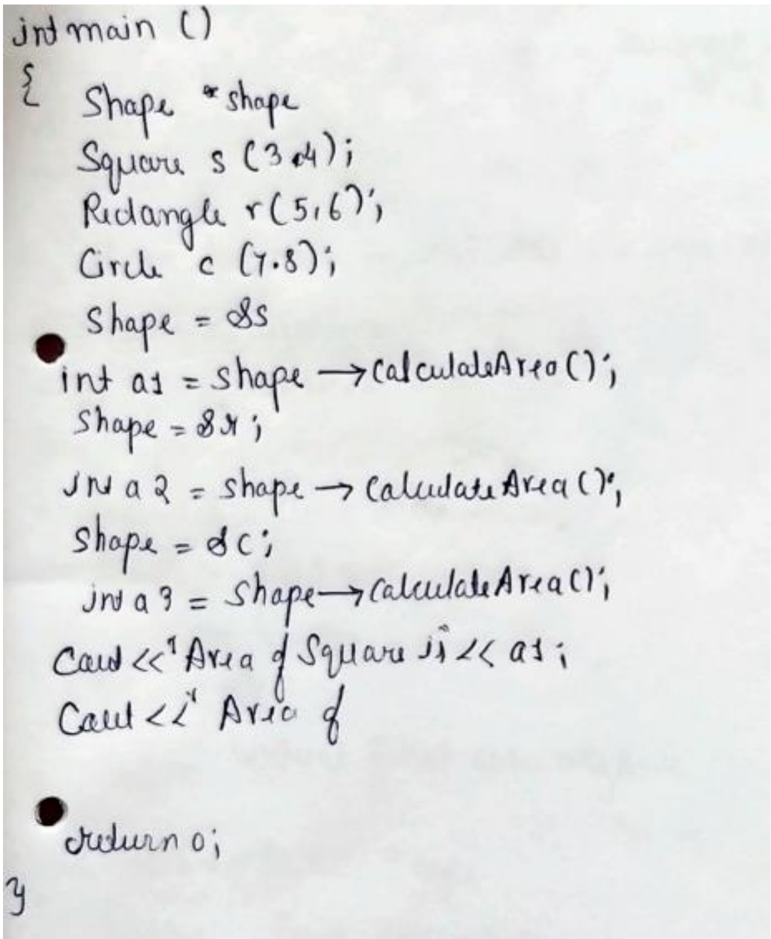
5 The $\boxed{=0}$ syntax does not mean we are assigning 0 to the function. Its just the way we define Pure virtual functions.



```
Class Person
{ Public: virtual void fun() = 0;
   void f1() {    }
}
Class Student : Public Person
{ Public; void fun()
          {
          }
};
```

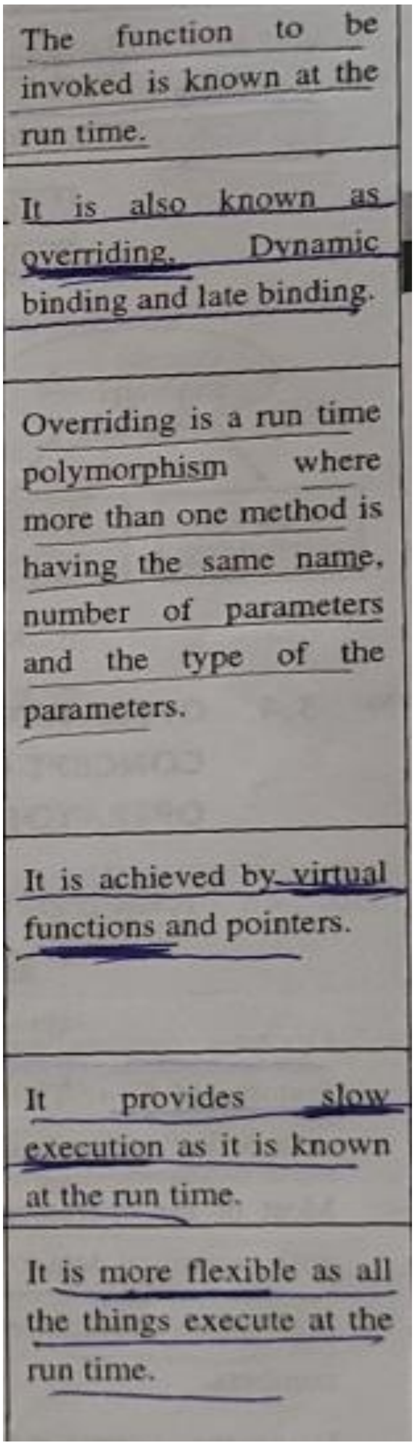


```
# include<iostream.h>
using namespace std;
 //Abstract class
Class shape
{
  Public : virtual float calculateArea() = 0; // Pure virtual
                                                     function
};
Class Square : Public Shape
{ float a;
  Public : Square (float l)
           { a = l;
           }
           float CalculateArea ()
            { return a*a;
};          }
Class Circle : Public Shape
{
  float r;
   Public : Circle (float x)
            { r = x; }
            }
            float calculateArea()
             {
               return 3.14 * r * r;
};           }
```



```
int main ()
{  Shape *shape
   Square s (3,4);
   Rectangle r(5,6);
   Circle c (7.8);
   Shape = &s
  int a1 = shape -> calculateArea();
   Shape = &r;
  int a2 = shape -> CalculateArea();
   shape = &c;
   int a3 = Shape -> calculateArea();
  Cout <<"Area of Square is"<< a1;
  Cout <<" Area of

  return 0;
}
```

4) What is runtime polymorphism? How it is achieved in C++. Explain it along with example



The function to be invoked is known at the run time.

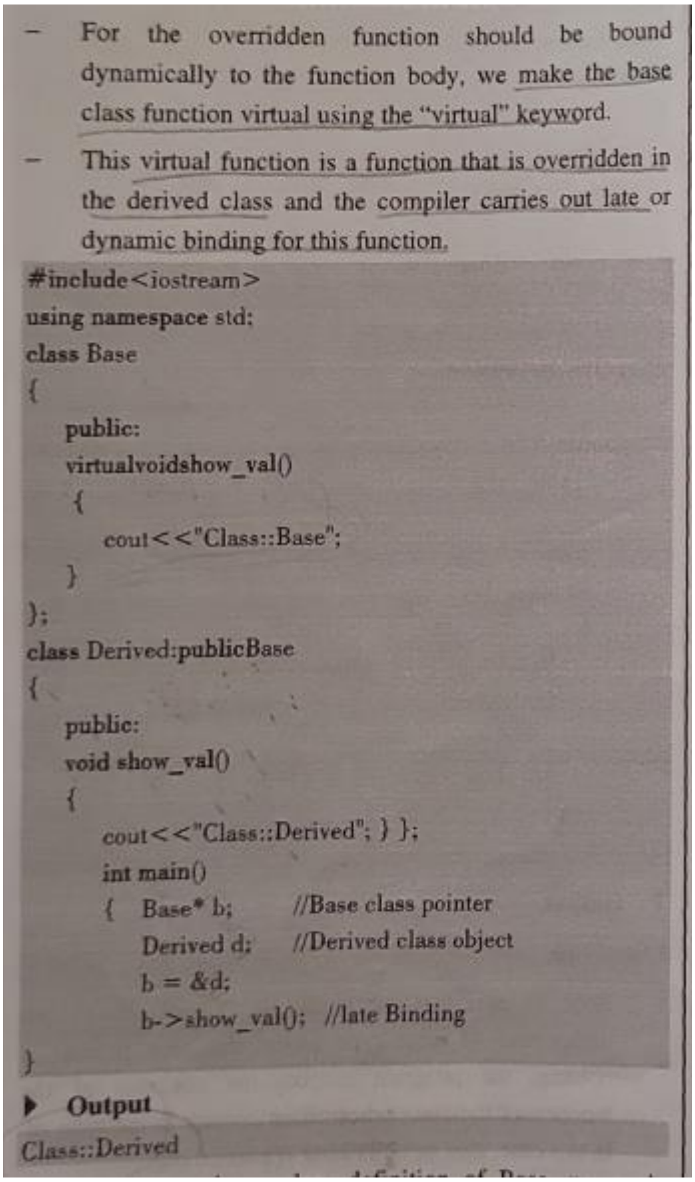
It is also known as overriding, Dynamic binding and late binding.

Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.

It is achieved by virtual functions and pointers.

It provides slow execution as it is known at the run time.

It is more flexible as all the things execute at the run time.



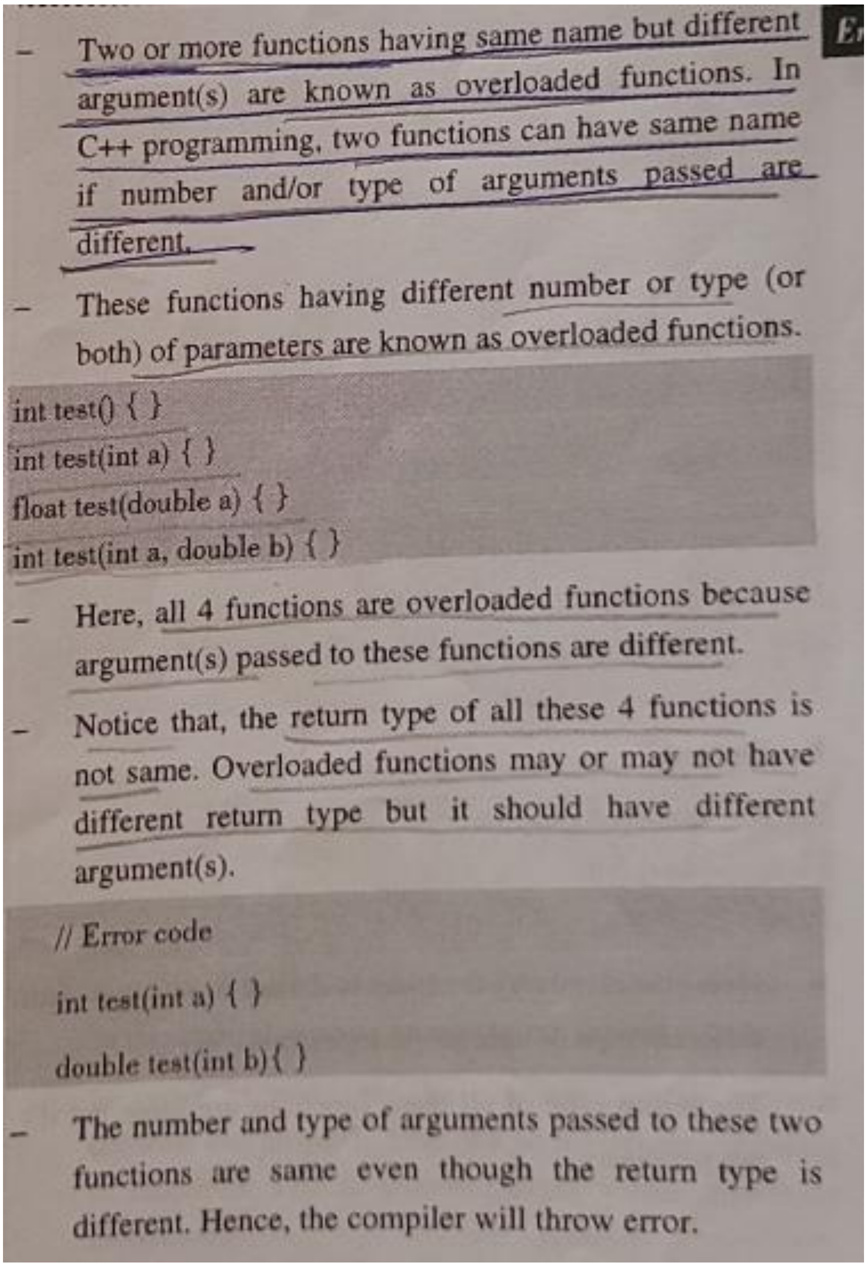
- For the overridden function should be bound dynamically to the function body, we make the base class function virtual using the "virtual" keyword.
- This virtual function is a function that is overridden in the derived class and the compiler carries out late or dynamic binding for this function.

```
#include<iostream>
using namespace std;
class Base
{
   public:
   virtualvoidshow_val()
    {
       cout<<"Class::Base";
    }
};
class Derived:publicBase
{
   public:
   void show_val()
   {
       cout<<"Class::Derived"; } };
       int main()
       {  Base* b;       //Base class pointer
          Derived d;     //Derived class object
          b = &d;
          b->show_val();  //late Binding
}
```

▶ **Output**

```
Class::Derived
```

5) **What is function overloading? Write defination of three overloaded functions (add) which will add two integer, float and double numbers respectively.**



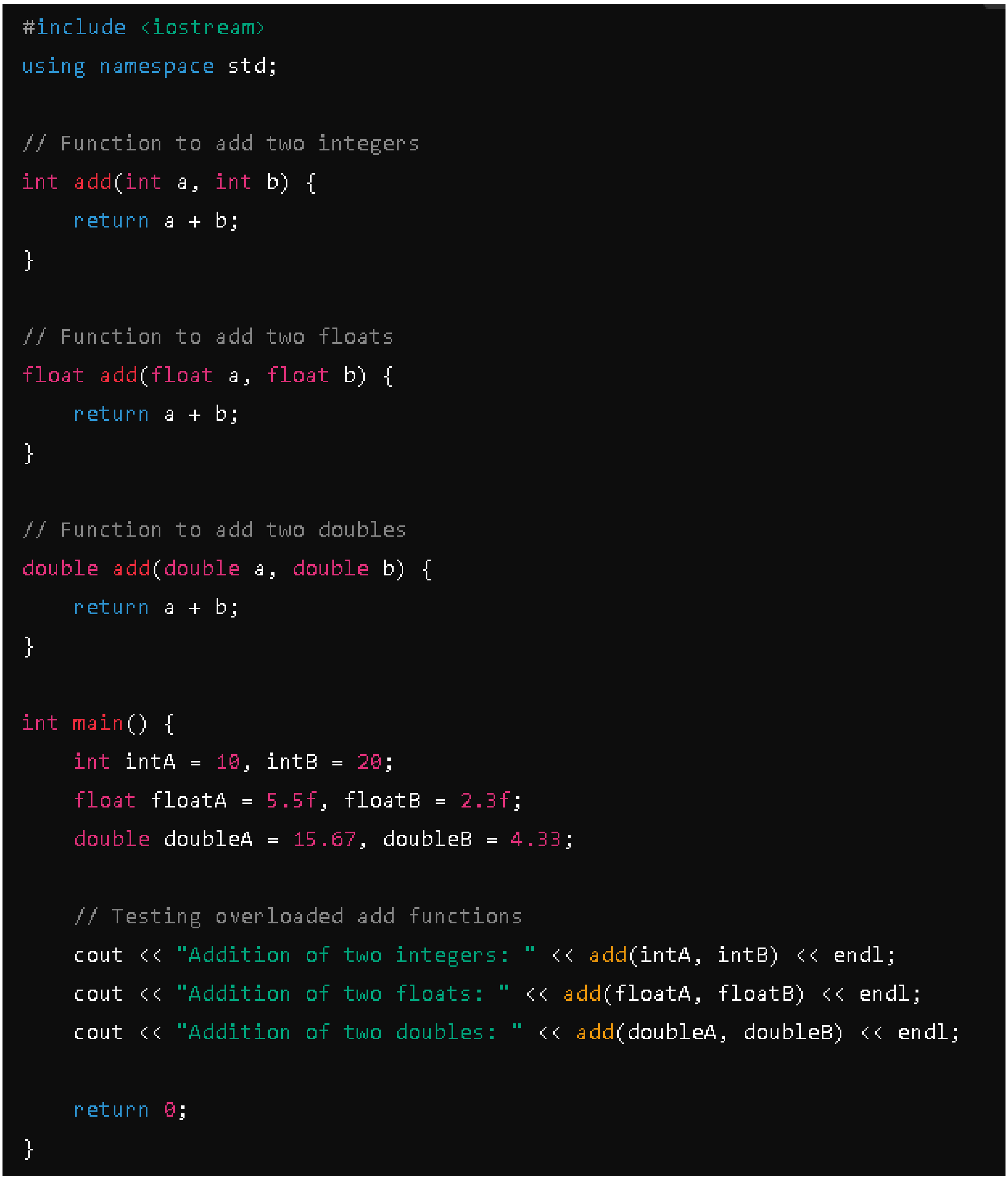
- Two or more functions having same name but different argument(s) are known as overloaded functions. In C++ programming, two functions can have same name if number and/or type of arguments passed are different.
- These functions having different number or type (or both) of parameters are known as overloaded functions.

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

- Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.
- Notice that, the return type of all these 4 functions is not same. Overloaded functions may or may not have different return type but it should have different argument(s).

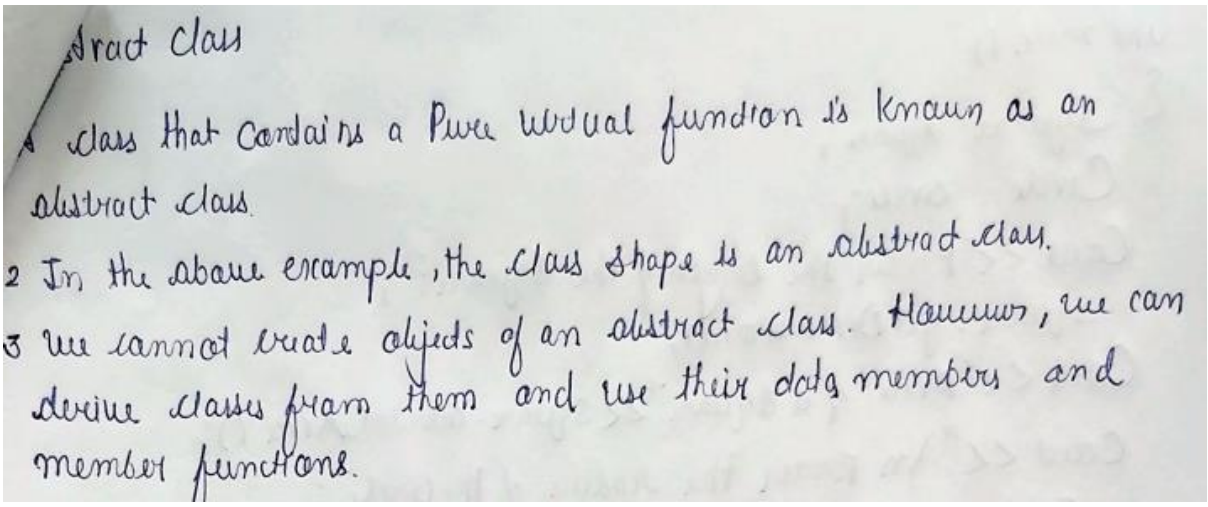
```
// Error code
int test(int a) { }
double test(int b){ }
```

- The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.



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

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Function to add two floats
float add(float a, float b) {
    return a + b;
}

// Function to add two doubles
double add(double a, double b) {
    return a + b;
}

int main() {
    int intA = 10, intB = 20;
    float floatA = 5.5f, floatB = 2.3f;
    double doubleA = 15.67, doubleB = 4.33;

    // Testing overloaded add functions
    cout << "Addition of two integers: " << add(intA, intB) << endl;
    cout << "Addition of two floats: " << add(floatA, floatB) << endl;
    cout << "Addition of two doubles: " << add(doubleA, doubleB) << endl;

    return 0;
}
```

6) Explain abstract class concept along with example.

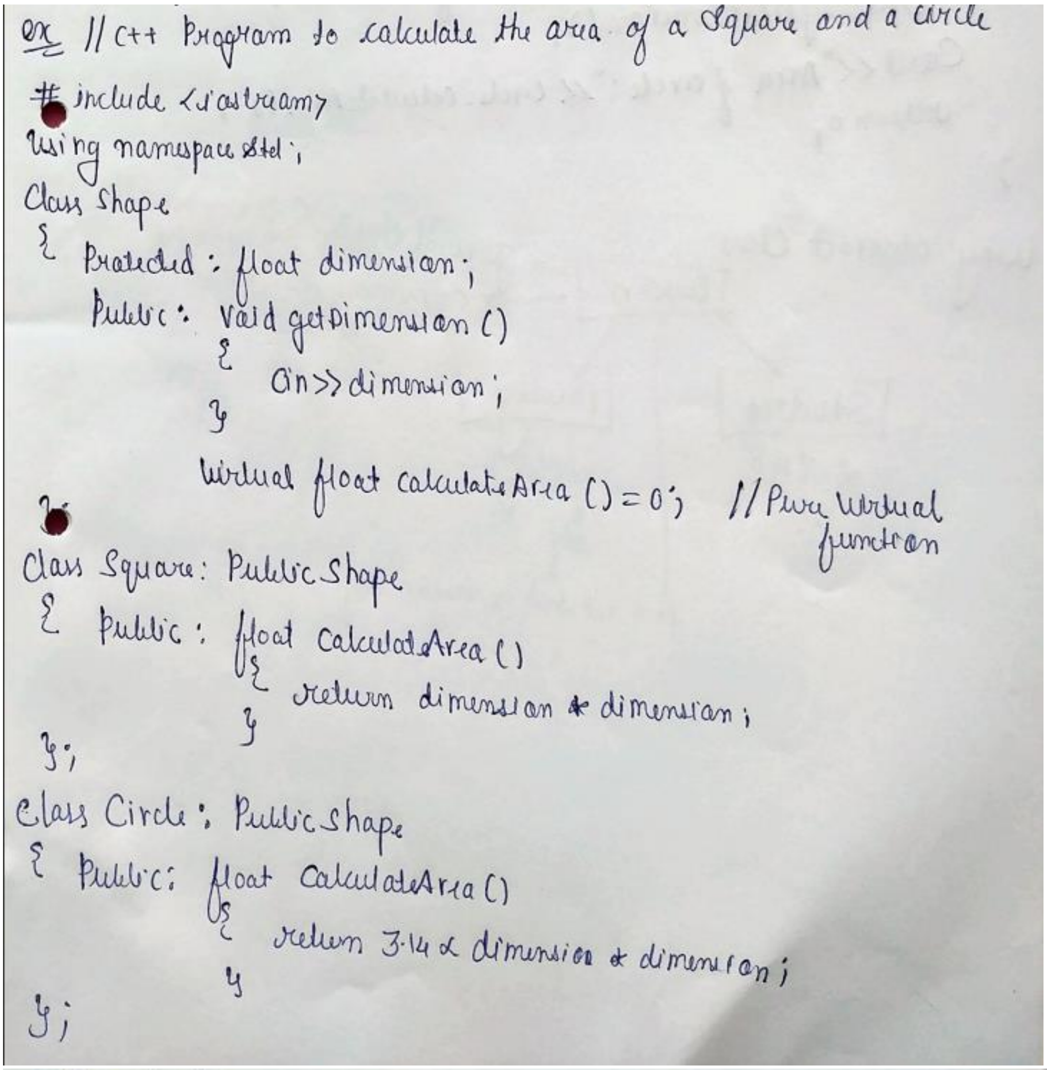


stract Class

A class that Contains a Pure virtual function is known as an abstract class.

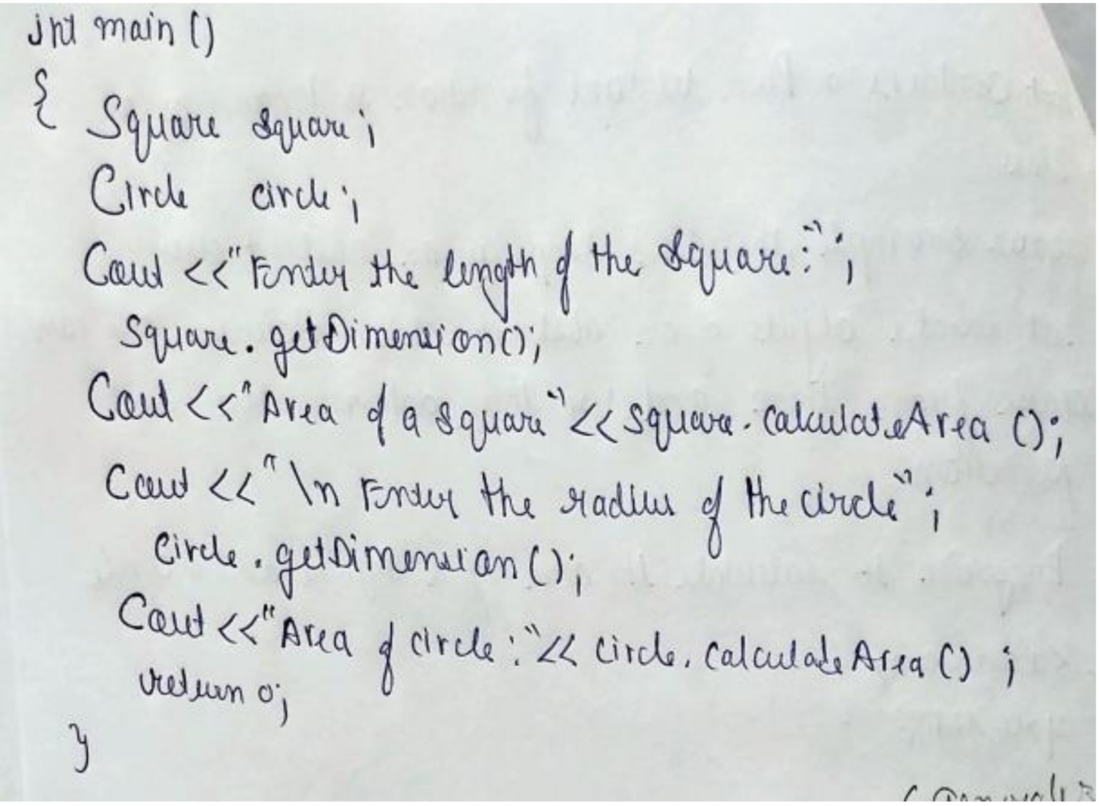
2 In the above example, the class Shape is an abstract class.

3 we cannot create objects of an abstract class. However, we can derive classes from them and use their data members and member functions.



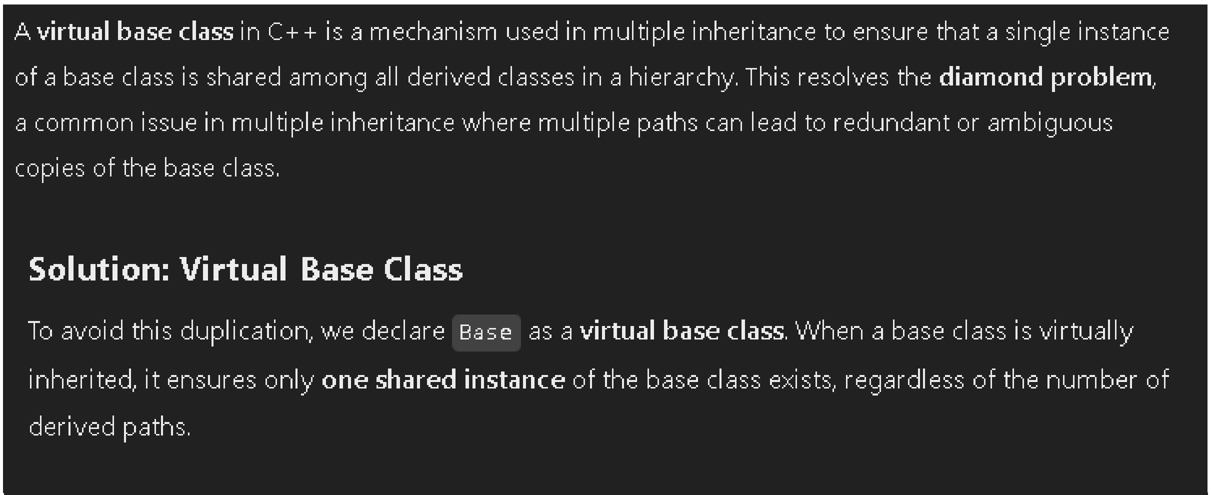
ex // C++ Program to calculate the area of a Square and a circle

```
# include <iostream>
using namespace std;
Class Shape
{ Protected : float dimension;
  Public : void getDimension ()
           {
              cin>> dimension;
           }
           virtual float calculateArea () = 0;   // Pure virtual
                                                    function
};
Class Square: Public Shape
{ Public : float CalculateArea ()
              {
                 return dimension * dimension;
              }
};
Class Circle : Public shape
{ Public: float CalculateArea ()
             {
                return 3.14 * dimension * dimension;
             }
};
```



```
int main ()
{ Square Square;
  Circle  circle;
  Cout <<"Enter the length of the Square:";
   Square. getDimension();
  Cout <<"Area of a Square" << Square. calculateArea ();
  Cout <<"\n Enter the radius of the circle";
    Circle . getDimension ();
  Cout <<"Area of circle :"<< Circle. calculateArea () ;
  return 0;
}
```

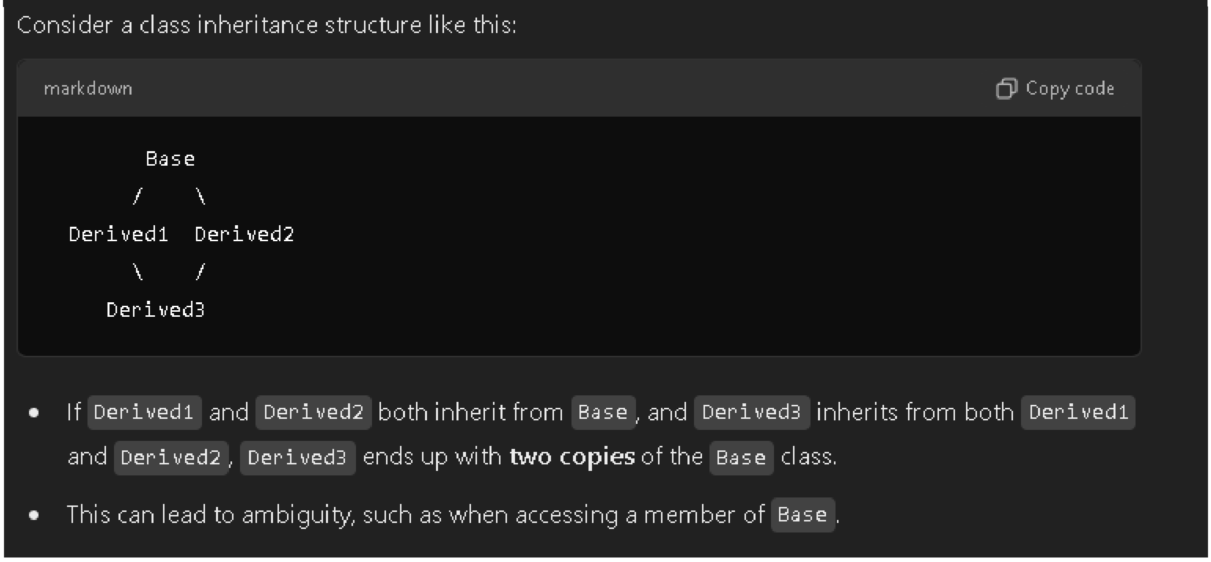
7) Explain virtual base class and virtual function with example



A **virtual base class** in C++ is a mechanism used in multiple inheritance to ensure that a single instance of a base class is shared among all derived classes in a hierarchy. This resolves the **diamond problem**, a common issue in multiple inheritance where multiple paths can lead to redundant or ambiguous copies of the base class.

Solution: Virtual Base Class

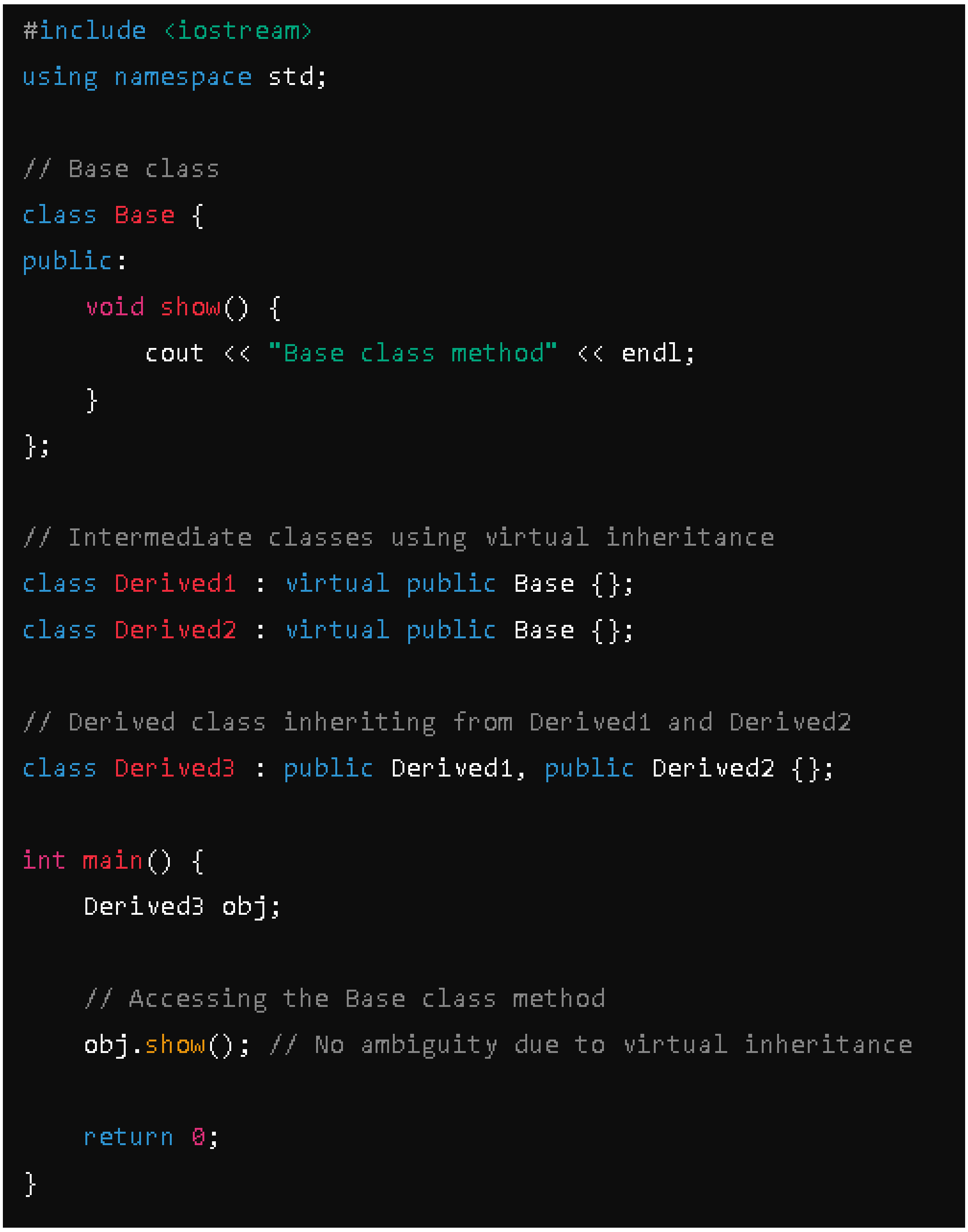
To avoid this duplication, we declare `Base` as a **virtual base class**. When a base class is virtually inherited, it ensures only **one shared instance** of the base class exists, regardless of the number of derived paths.



Consider a class inheritance structure like this:

```markdown
        Base
       /    \
  Derived1  Derived2
       \    /
     Derived3
```

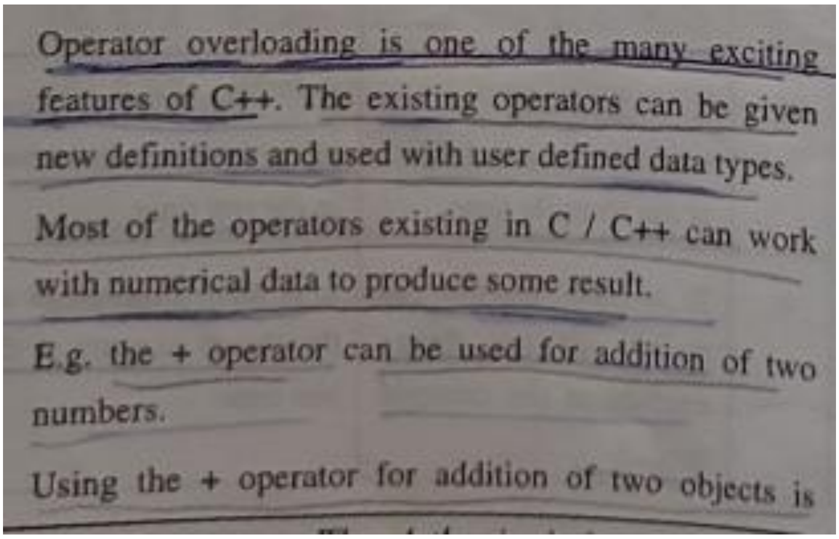
- If `Derived1` and `Derived2` both inherit from `Base`, and `Derived3` inherits from both `Derived1` and `Derived2`, `Derived3` ends up with **two copies** of the `Base` class.
- This can lead to ambiguity, such as when accessing a member of `Base`.



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

// Base class
class Base {
public:
    void show() {
        cout << "Base class method" << endl;
    }
};

// Intermediate classes using virtual inheritance
class Derived1 : virtual public Base {};
class Derived2 : virtual public Base {};

// Derived class inheriting from Derived1 and Derived2
class Derived3 : public Derived1, public Derived2 {};

int main() {
    Derived3 obj;

    // Accessing the Base class method
    obj.show(); // No ambiguity due to virtual inheritance

    return 0;
}
```

FOR **VIRTUAL FUNCTION** PLEASE REFER **QUESTION NO. 4**

8) Explain need of operator overloading. Write C++ program to demonstrate use of unary operator overloading.

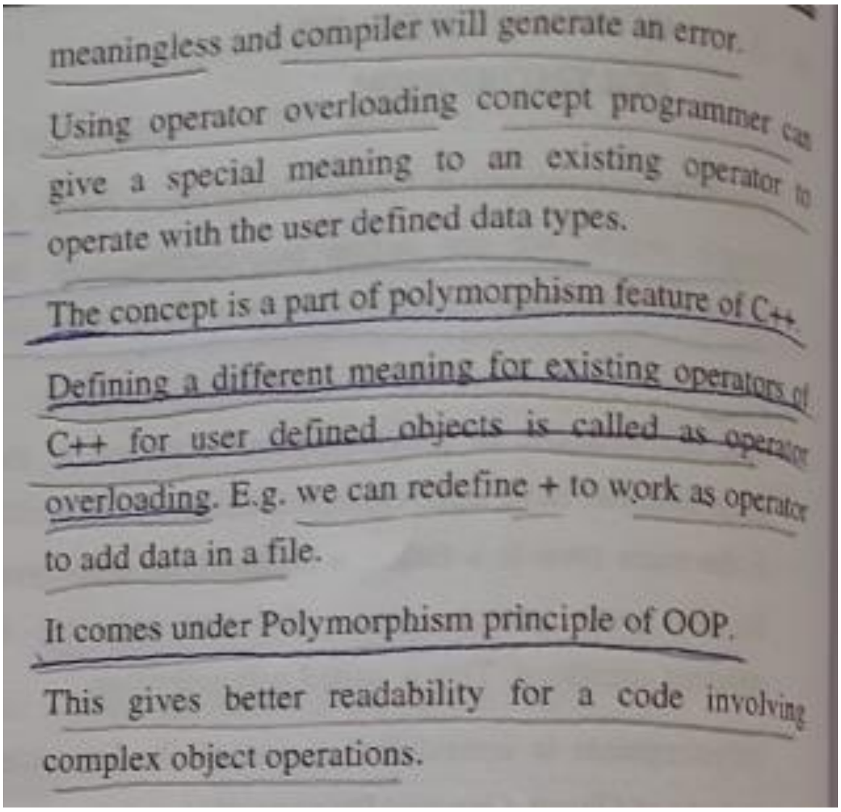


Operator overloading is one of the many exciting features of C++. The existing operators can be given new definitions and used with user defined data types.

Most of the operators existing in C / C++ can work with numerical data to produce some result.

E.g. the + operator can be used for addition of two numbers.

Using the + operator for addition of two objects is



meaningless and compiler will generate an error.

Using operator overloading concept programmer can give a special meaning to an existing operator to operate with the user defined data types.

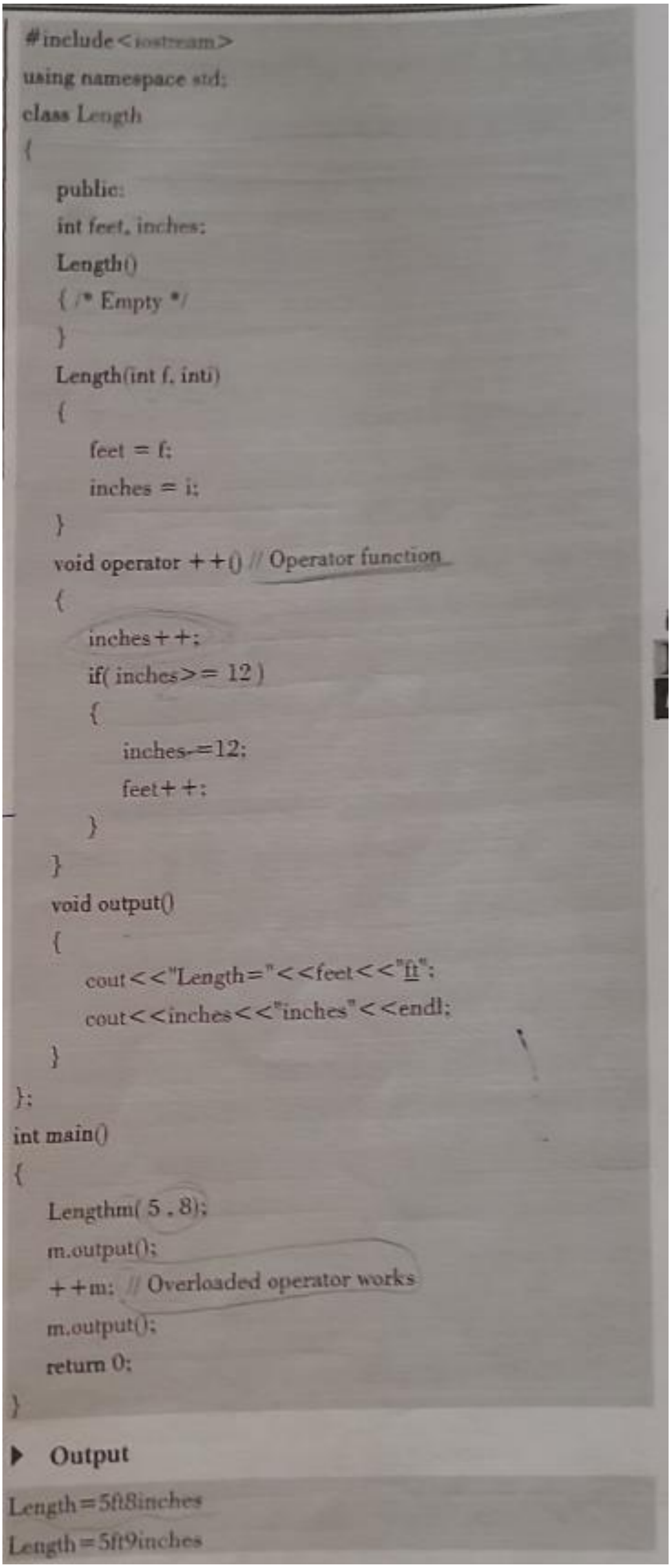
The concept is a part of polymorphism feature of C++.

Defining a different meaning for existing operators of C++ for user defined objects is called as operator overloading. E.g. we can redefine + to work as operator to add data in a file.

It comes under Polymorphism principle of OOP.

This gives better readability for a code involving complex object operations.

Overloading __ operator to increment length of an object by one inch

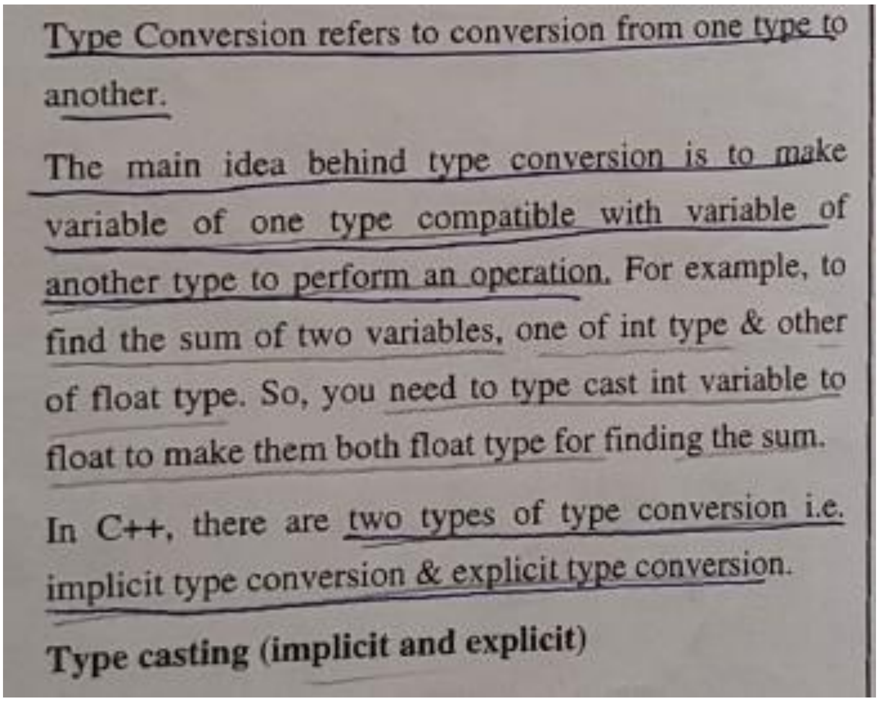


```
#include<iostream>
using namespace std;
class Length
{
    public:
    int feet, inches;
    Length()
    { /* Empty */
    }
    Length(int f, inti)
    {
        feet = f;
        inches = i;
    }
    void operator ++() // Operator function
    {
        inches++;
        if( inches>= 12 )
        {
            inches-=12;
            feet++;
        }
    }
    void output()
    {
        cout<<"Length="<<feet<<"ft";
        cout<<inches<<"inches"<<endl;
    }
};
int main()
{
    Lengthm( 5 , 8);
    m.output();
    ++m; // Overloaded operator works
    m.output();
    return 0;
}
```

▶ **Output**

```
Length=5ft8inches
Length=5ft9inches
```

9) Explain what is type casting, Explain Implicit and explicit type of conversion with example

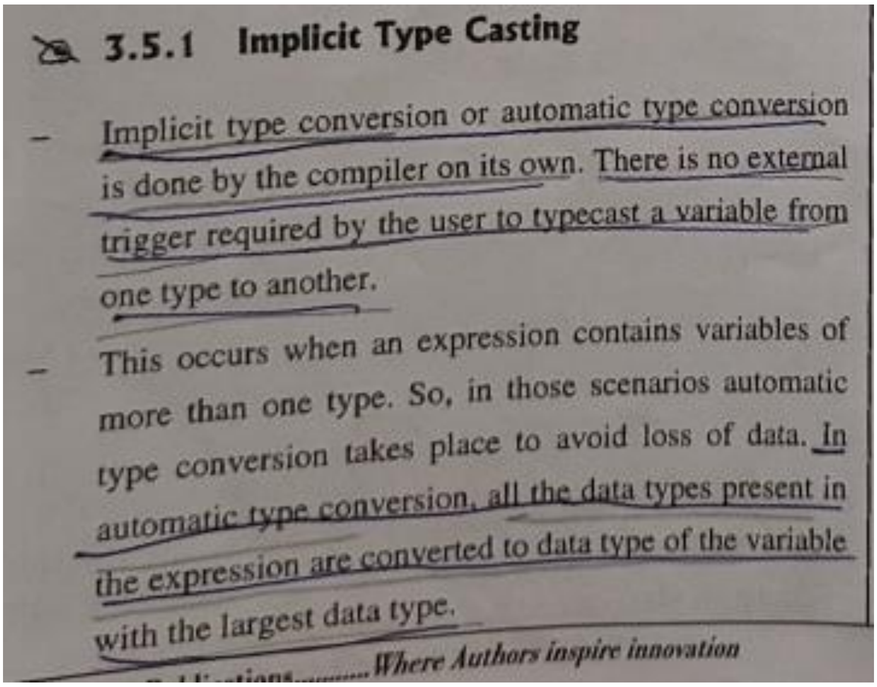


Type Conversion refers to conversion from one type to another.

The main idea behind type conversion is to make variable of one type compatible with variable of another type to perform an operation. For example, to find the sum of two variables, one of int type & other of float type. So, you need to type cast int variable to float to make them both float type for finding the sum.

In C++, there are two types of type conversion i.e. implicit type conversion & explicit type conversion.

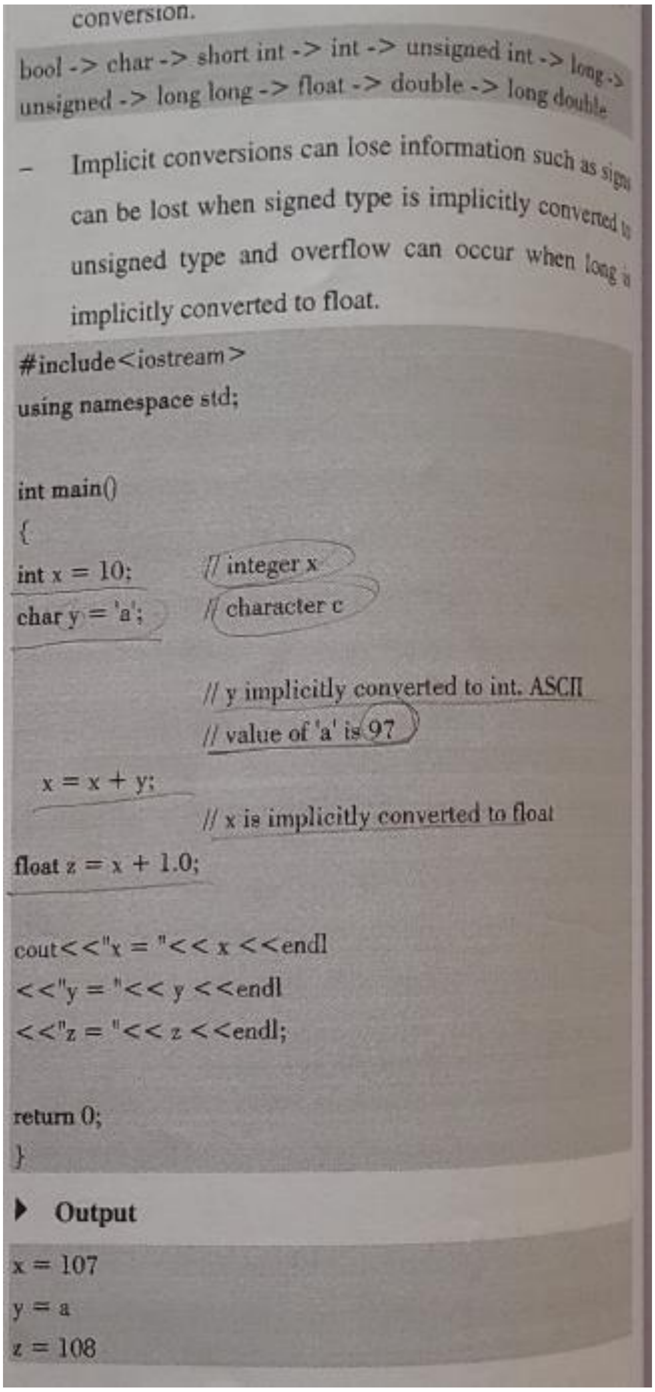
Type casting (implicit and explicit)



3.5.1 Implicit Type Casting

- Implicit type conversion or automatic type conversion is done by the compiler on its own. There is no external trigger required by the user to typecast a variable from one type to another.
- This occurs when an expression contains variables of more than one type. So, in those scenarios automatic type conversion takes place to avoid loss of data. In automatic type conversion, all the data types present in the expression are converted to data type of the variable with the largest data type.

IMPLICIT TYPE CASTING EXAMPLE



conversion.

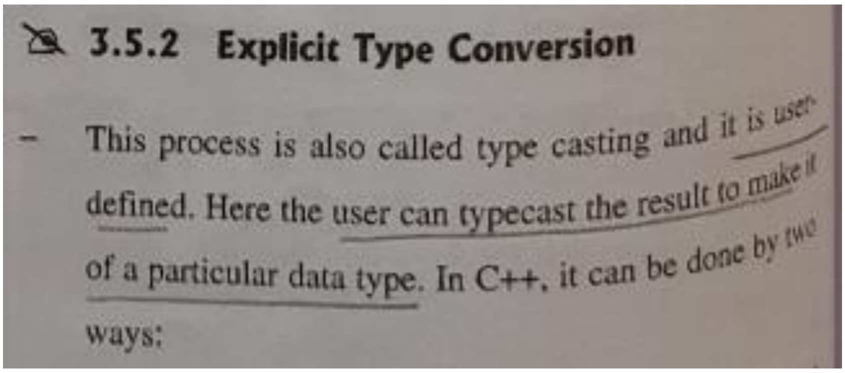
bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

- Implicit conversions can lose information such as signs can be lost when signed type is implicitly converted to unsigned type and overflow can occur when long is implicitly converted to float.

```
#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;
}
```

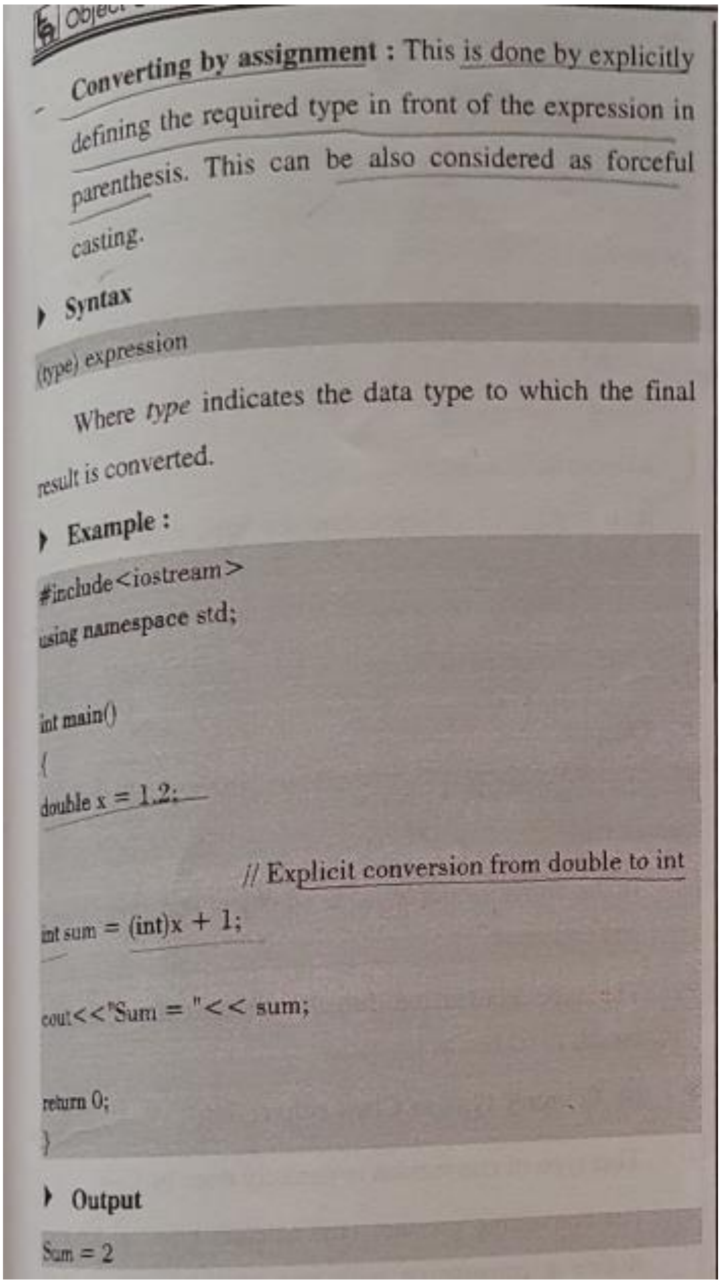
▶ **Output**

```
x = 107
y = a
z = 108
```



3.5.2 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.

Syntax

```
(type) expression
```

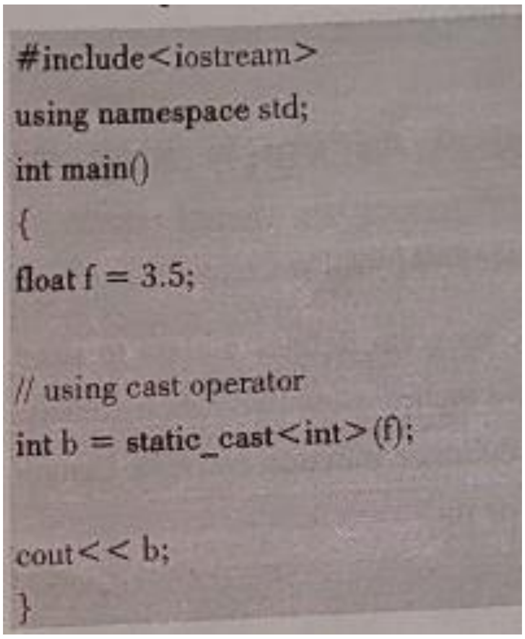
Where *type* indicates the data type to which the final result is converted.

Example :

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

int main()
{
double x = 1.2;

                    // Explicit conversion from double to int

int sum = (int)x + 1;

cout<<"Sum = "<< sum;

return 0;
}
```

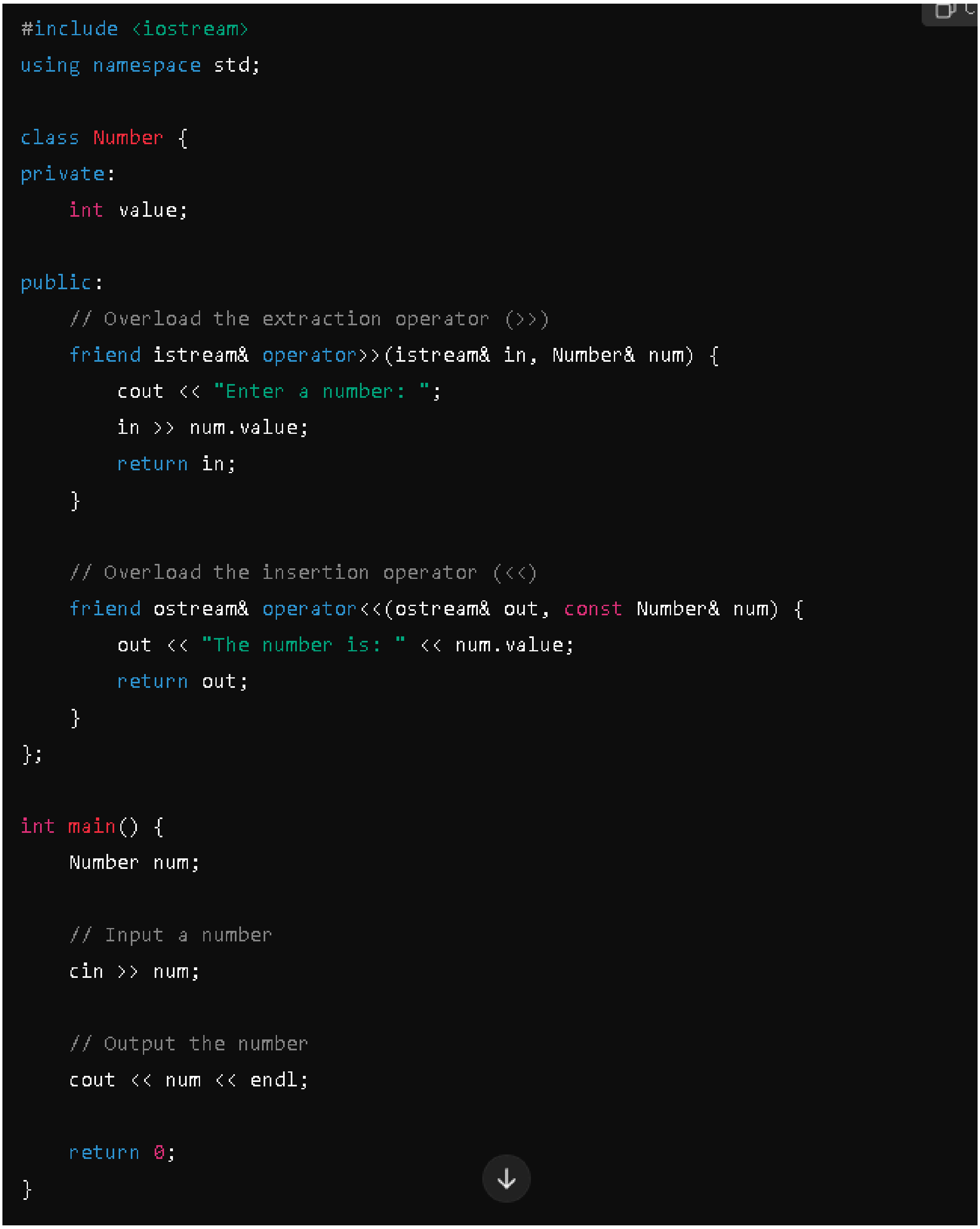
Output

```
Sum = 2
```



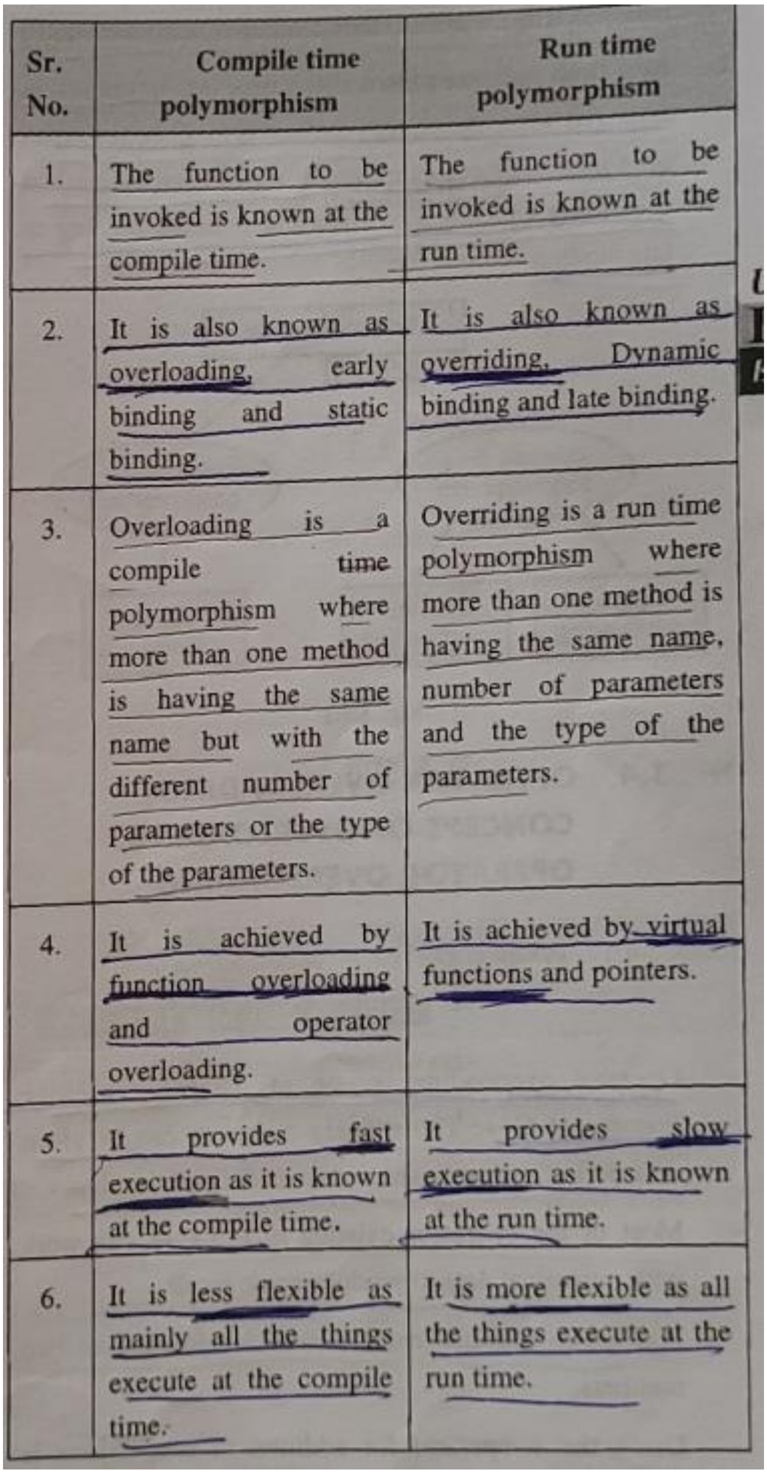
```
#include<iostream>
using namespace std;
int main()
{
float f = 3.5;

// using cast operator
int b = static_cast<int>(f);

cout<< b;
}
```

10) Write a program to overload insertion (<<) and extraction (>>) operator in C++



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

class Number {
private:
    int value;

public:
    // Overload the extraction operator (>>)
    friend istream& operator>>(istream& in, Number& num) {
        cout << "Enter a number: ";
        in >> num.value;
        return in;
    }

    // Overload the insertion operator (<<)
    friend ostream& operator<<(ostream& out, const Number& num) {
        out << "The number is: " << num.value;
        return out;
    }
};

int main() {
    Number num;

    // Input a number
    cin >> num;

    // Output the number
    cout << num << endl;

    return 0;
}
```

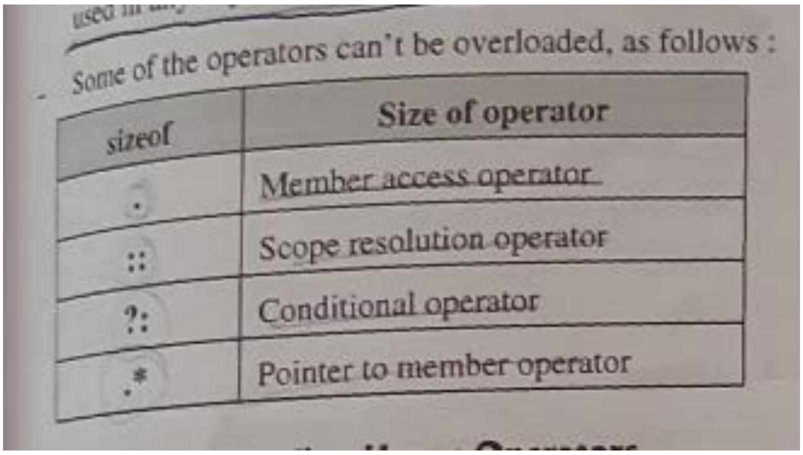
11) Differentiate between compile time polymorphism and run time polymorphism



Sr. No.	Compile time polymorphism	Run time polymorphism
1.	The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
2.	It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
3.	Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
4.	It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
5.	It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
6.	It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

12) What is operator overloading and why it is useful? Which Operators cannot be overloaded

Refer Question No 8

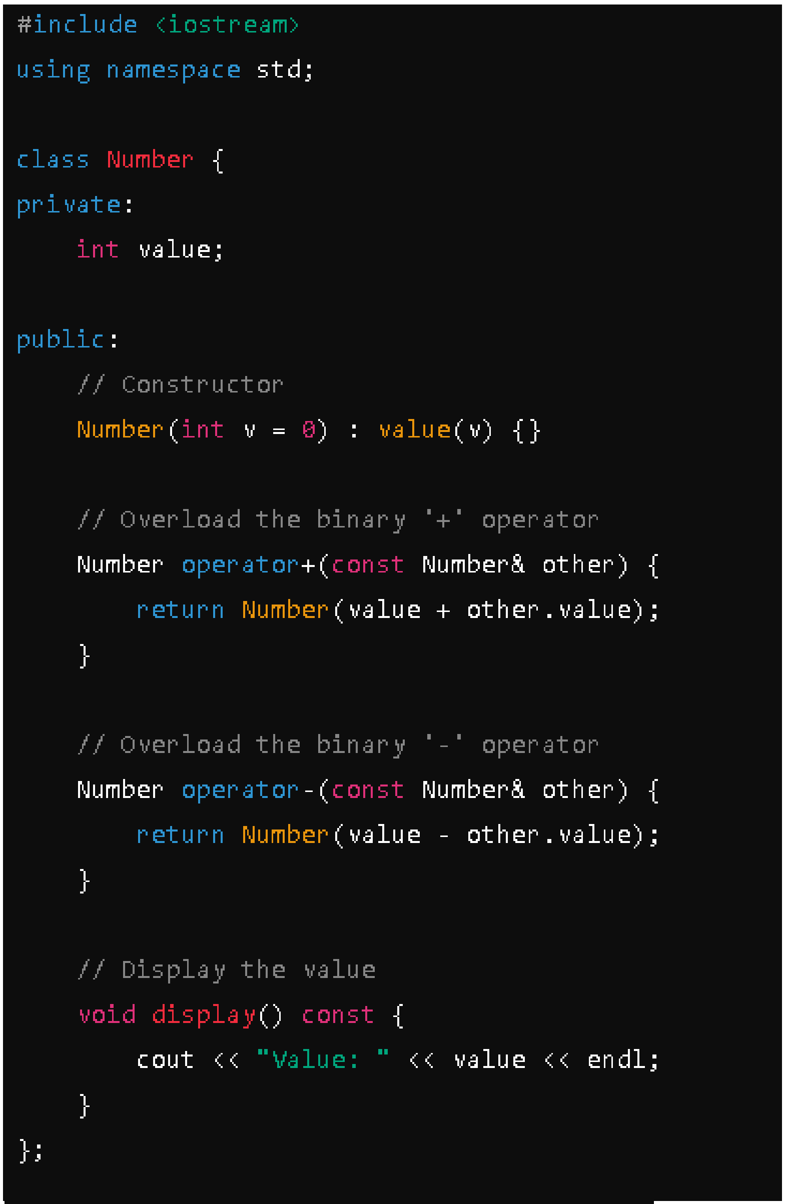


used in any

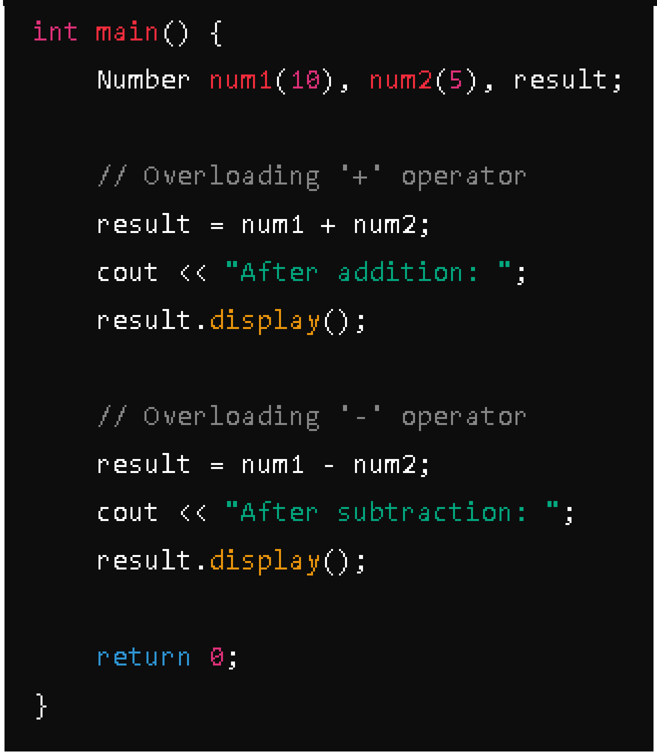
Some of the operators can't be overloaded, as follows :

sizeof	Size of operator
.	Member access operator
::	Scope resolution operator
?:	Conditional operator
.*	Pointer to member operator

13) **Write a program to binary (+) and binary (-) operator in C++**



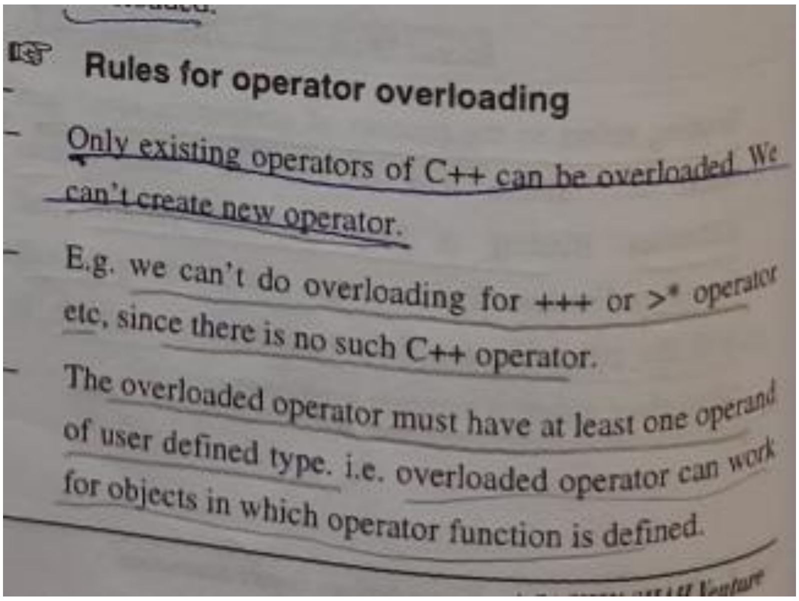
```cpp
#include <iostream>
using namespace std;

class Number {
private:
    int value;

public:
    // Constructor
    Number(int v = 0) : value(v) {}

    // Overload the binary '+' operator
    Number operator+(const Number& other) {
        return Number(value + other.value);
    }

    // Overload the binary '-' operator
    Number operator-(const Number& other) {
        return Number(value - other.value);
    }

    // Display the value
    void display() const {
        cout << "Value: " << value << endl;
    }
};
```



```cpp
int main() {
    Number num1(10), num2(5), result;

    // Overloading '+' operator
    result = num1 + num2;
    cout << "After addition: ";
    result.display();

    // Overloading '-' operator
    result = num1 - num2;
    cout << "After subtraction: ";
    result.display();

    return 0;
}
```

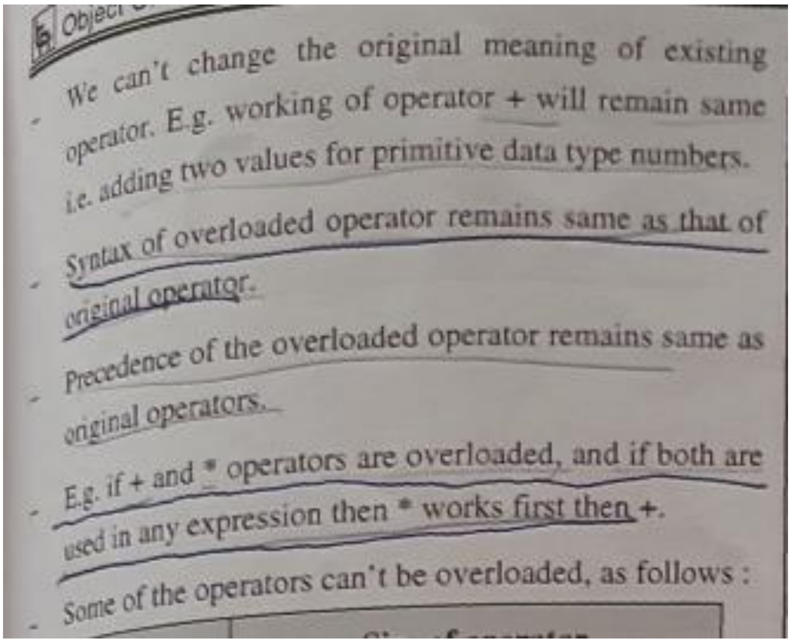
14) **Explain need of operator overloading. What are the rules to be followed when overloading an operator in C++?**

Refer Question No. 8



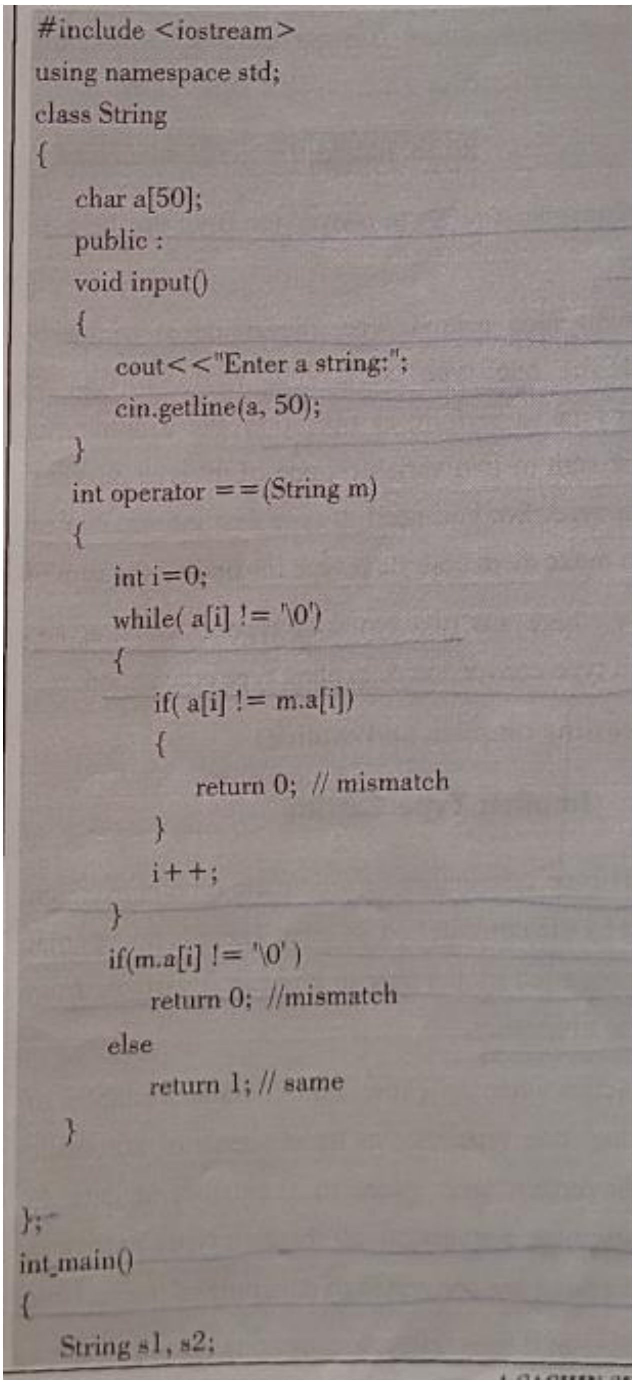
Rules for operator overloading

- Only existing operators of C++ can be overloaded. We can't create new operator.
- E.g. we can't do overloading for +++ or >* operator etc, since there is no such C++ operator.
- The overloaded operator must have at least one operand of user defined type. i.e. overloaded operator can work for objects in which operator function is defined.

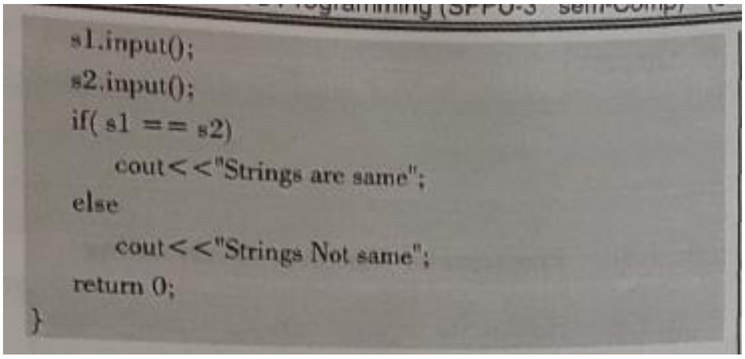


- We can't change the original meaning of existing operator. E.g. working of operator + will remain same i.e. adding two values for primitive data type numbers.
- Syntax of overloaded operator remains same as that of original operator.
- Precedence of the overloaded operator remains same as original operators.
- E.g. if + and * operators are overloaded, and if both are used in any expression then * works first then +.
- Some of the operators can't be overloaded, as follows :

15) **Define a class string and use binary overloaded operator (= =) to compare two strings**

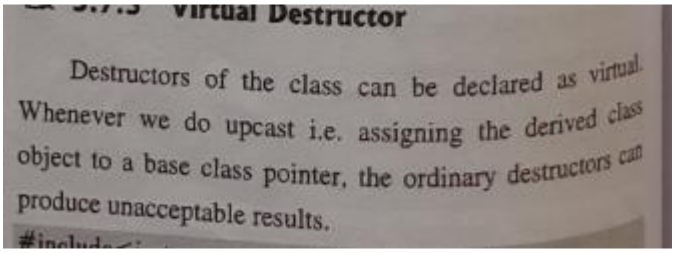


```
#include <iostream>
using namespace std;
class String
{
    char a[50];
    public :
    void input()
    {
        cout<<"Enter a string:";
        cin.getline(a, 50);
    }
    int operator ==(String m)
    {
        int i=0;
        while( a[i] != '\0')
        {
            if( a[i] != m.a[i])
            {
                return 0;  // mismatch
            }
            i++;
        }
        if(m.a[i] != '\0' )
            return 0;  //mismatch
        else
            return 1; // same
    }

};
int main()
{
    String s1, s2;
```



```
    s1.input();
    s2.input();
    if( s1 == s2)
        cout<<"Strings are same";
    else
        cout<<"Strings Not same";
    return 0;
}
```

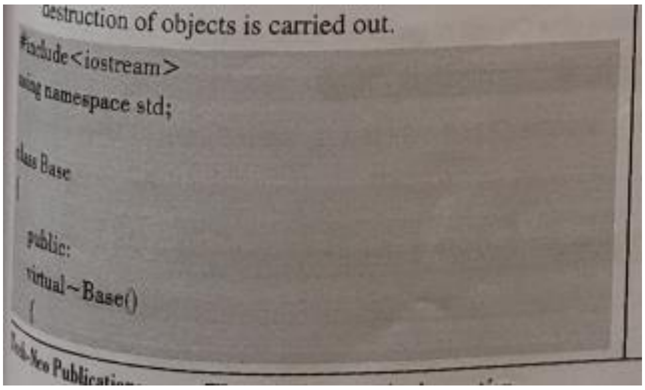
16) Explain Virtual destructor with the help of a program.



5.7.3 Virtual Destructor

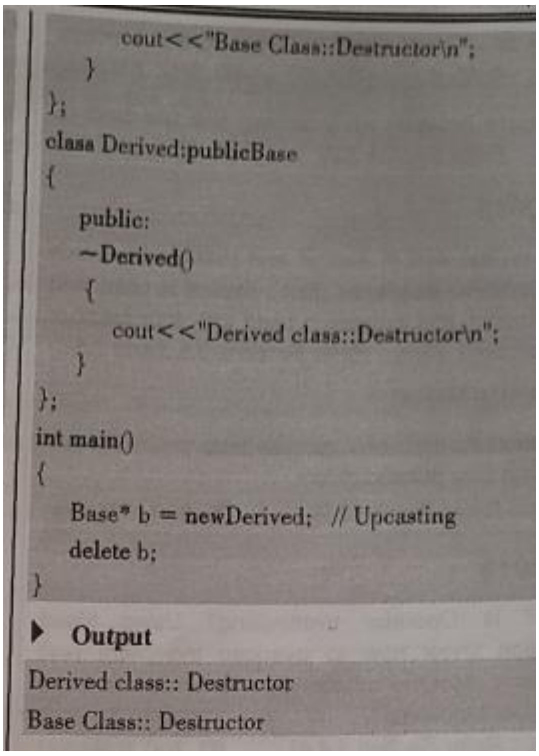
Destructors of the class can be declared as virtual. Whenever we do upcast i.e. assigning the derived class object to a base class pointer, the ordinary destructors can produce unacceptable results.

```
#include<i
```



destruction of objects is carried out.

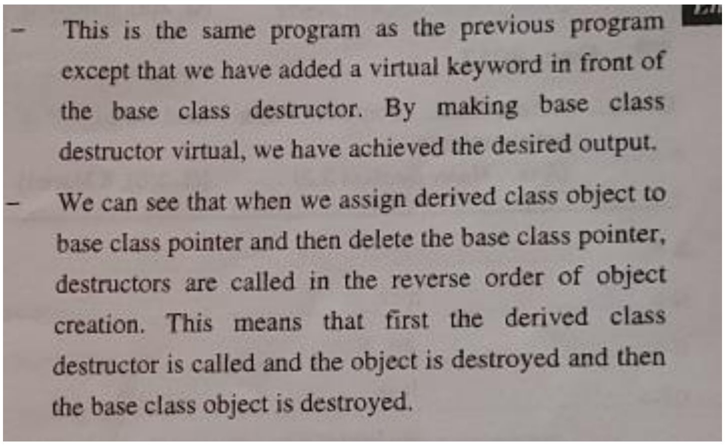
```
#include<iostream>
using namespace std;

class Base
{

public:
virtual~Base()
 {
```



```
        cout<<"Base Class::Destructor\n";
    }
};
class Derived:publicBase
{
    public:
    ~Derived()
    {
        cout<<"Derived class::Destructor\n";
    }
};
int main()
{
    Base* b = newDerived;  // Upcasting
    delete b;
}
```

▶ **Output**

```
Derived class:: Destructor
Base Class:: Destructor
```

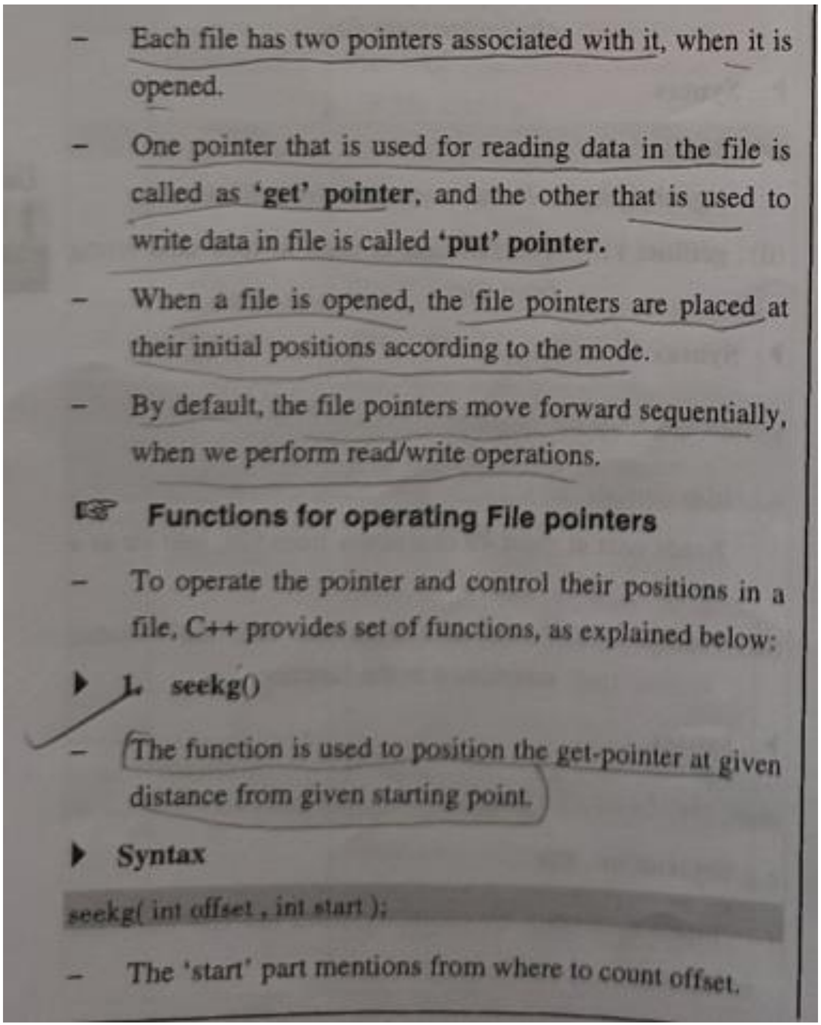


- This is the same program as the previous program except that we have added a virtual keyword in front of the base class destructor. By making base class destructor virtual, we have achieved the desired output.
- We can see that when we assign derived class object to base class pointer and then delete the base class pointer, destructors are called in the reverse order of object creation. This means that first the derived class destructor is called and the object is destroyed and then the base class object is destroyed.

UNIT : 4

FILES AND STREAMS

1) **What are various functions used to manipulate file pointers? Explain using example.**



- Each file has two pointers associated with it, when it is opened.
- One pointer that is used for reading data in the file is called as **'get' pointer**, and the other that is used to write data in file is called **'put' pointer.**
- When a file is opened, the file pointers are placed at their initial positions according to the mode.
- By default, the file pointers move forward sequentially, when we perform read/write operations.

☞ Functions for operating File pointers

- To operate the pointer and control their positions in a file, C++ provides set of functions, as explained below:

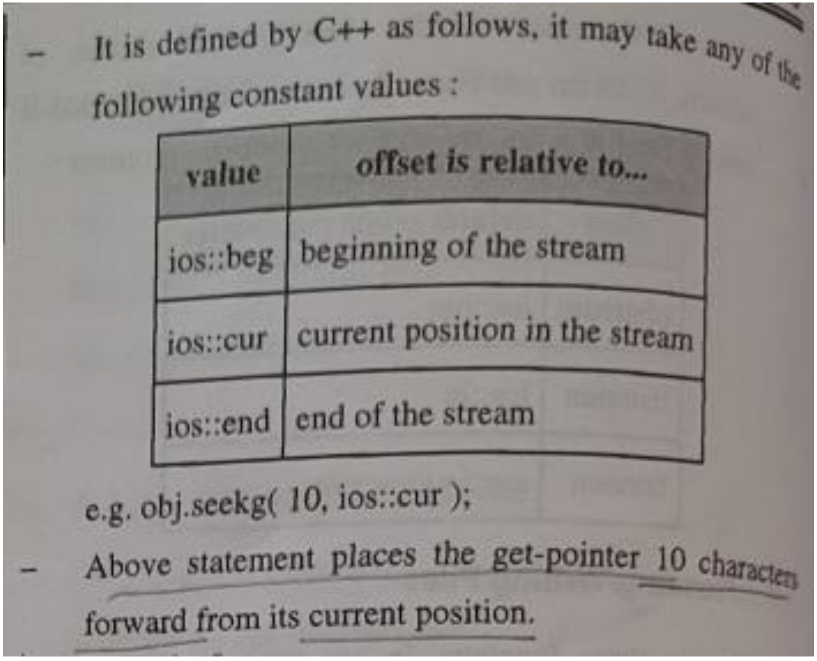
▶ **1. seekg()**

- The function is used to position the get-pointer at given distance from given starting point.

▶ **Syntax**

```
seekg( int offset , int start );
```

- The 'start' part mentions from where to count offset.

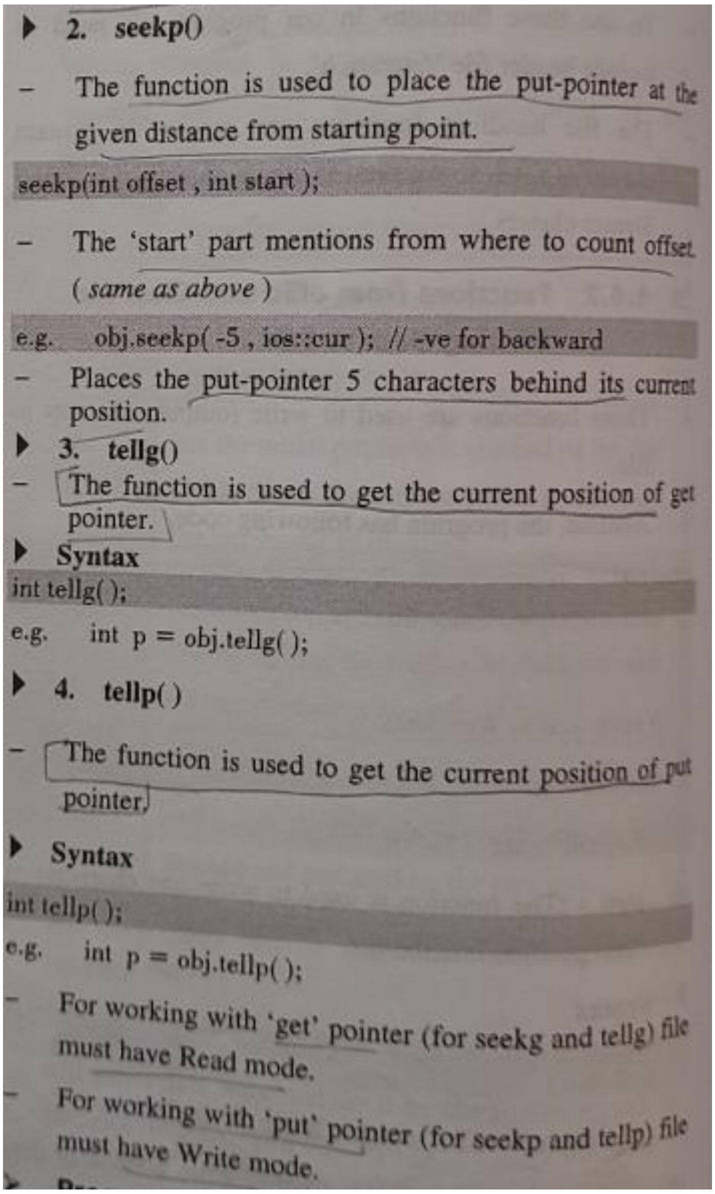


- It is defined by C++ as follows, it may take any of the following constant values :

value	offset is relative to...
ios::beg	beginning of the stream
ios::cur	current position in the stream
ios::end	end of the stream

e.g. obj.seekg(10, ios::cur);

- Above statement places the get-pointer 10 characters forward from its current position.



2. seekp()

- The function is used to place the put-pointer at the given distance from starting point.

```
seekp(int offset , int start );
```

- The 'start' part mentions from where to count offset. (*same as above*)

```
e.g.   obj.seekp( -5 , ios::cur );  // -ve for backward
```

- Places the put-pointer 5 characters behind its current position.

3. tellg()

- The function is used to get the current position of get pointer.

Syntax

```
int tellg( );
```

```
e.g.   int p = obj.tellg( );
```

4. tellp( )

- The function is used to get the current position of put pointer.

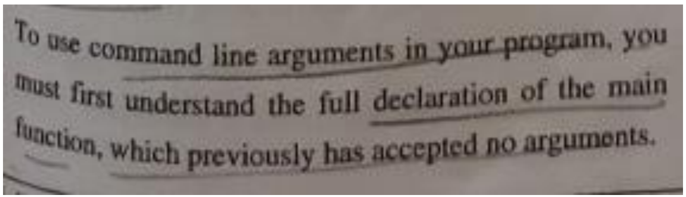
Syntax

```
int tellp( );
```

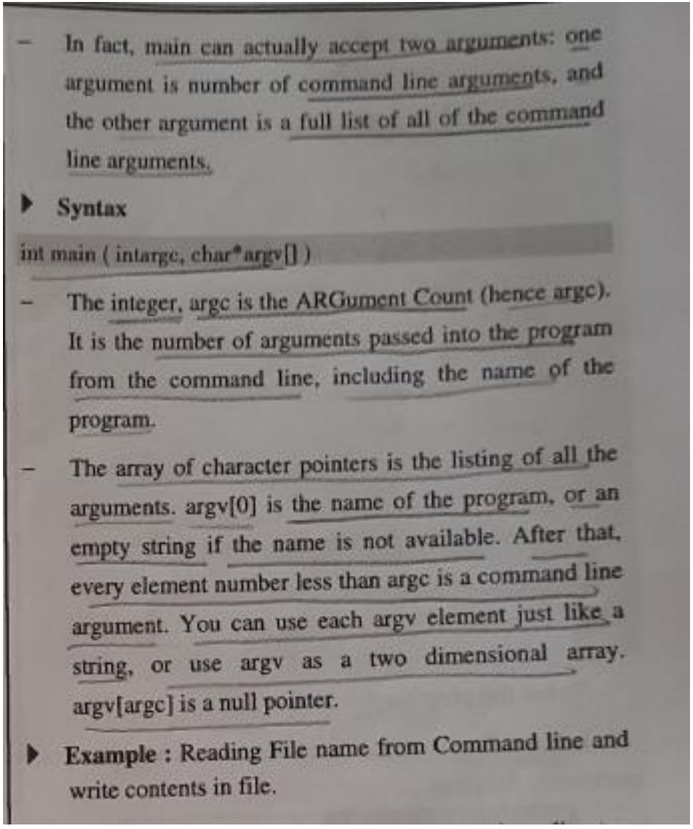
```
e.g.   int p = obj.tellp( );
```

- For working with 'get' pointer (for seekg and tellg) file must have Read mode.
- For working with 'put' pointer (for seekp and tellp) file must have Write mode.

2) What are command line arguments in C++? Write a program to explain the same



To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments.



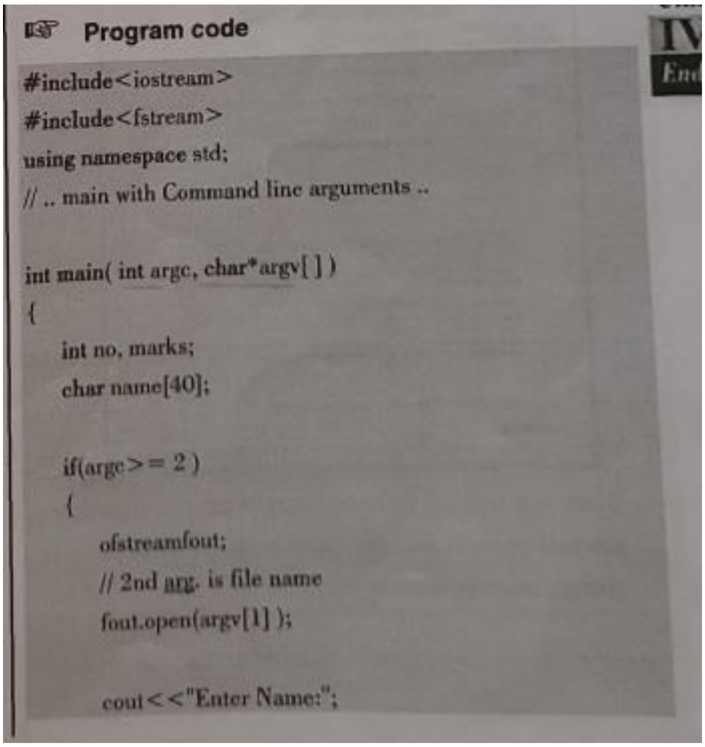
- In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

▶ **Syntax**

```
int main ( intargc, char*argv[] )
```

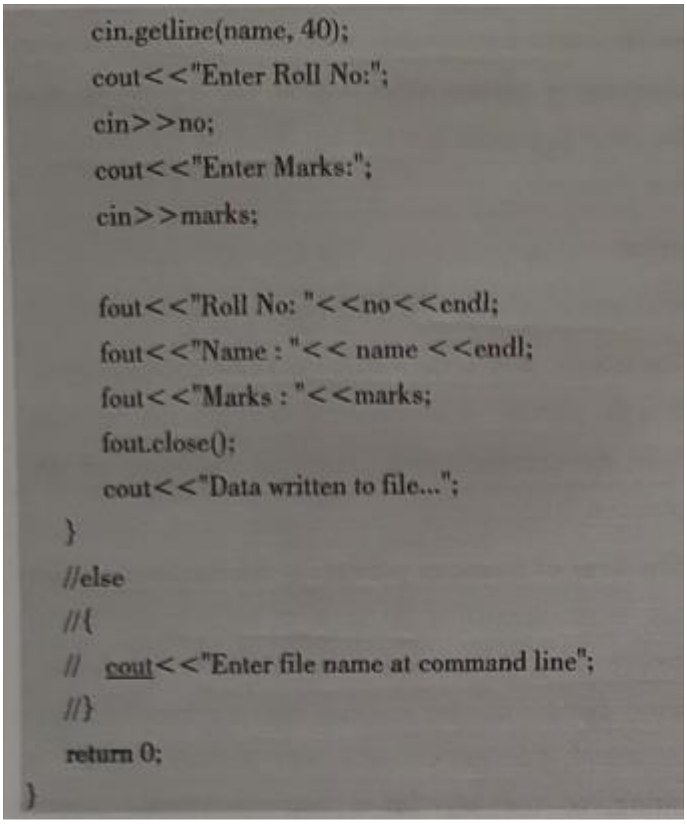
- The integer, argc is the ARGument Count (hence argc). It is the number of arguments passed into the program from the command line, including the name of the program.
- The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc is a command line argument. You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.

▶ **Example :** Reading File name from Command line and write contents in file.



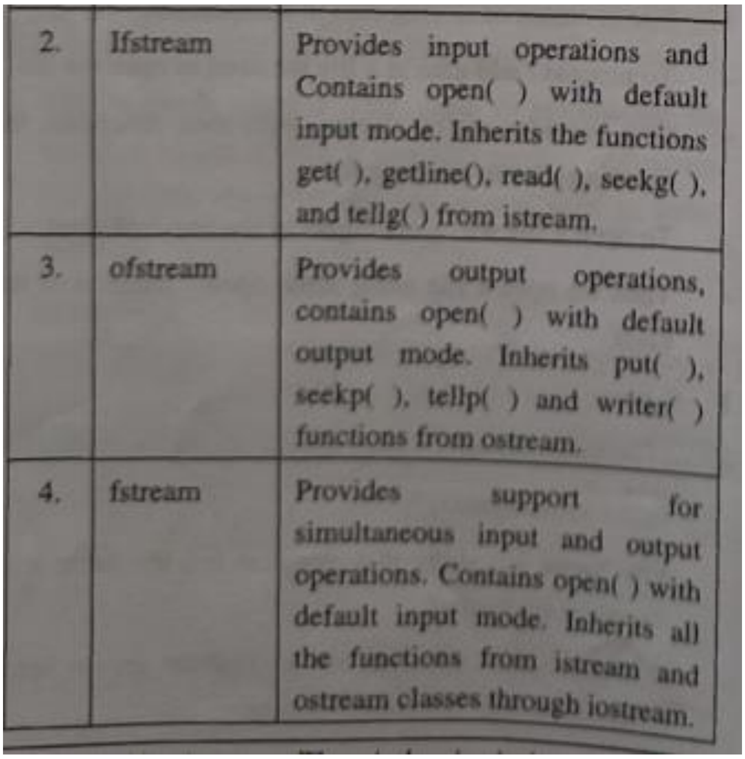
☞ **Program code**

```
#include<iostream>
#include<fstream>
using namespace std;
// .. main with Command line arguments ..

int main( int argc, char*argv[ ] )
{
    int no, marks;
    char name[40];

    if(argc>= 2 )
    {
        ofstreamfout;
        // 2nd arg. is file name
        fout.open(argv[1] );

        cout<<"Enter Name:";
```

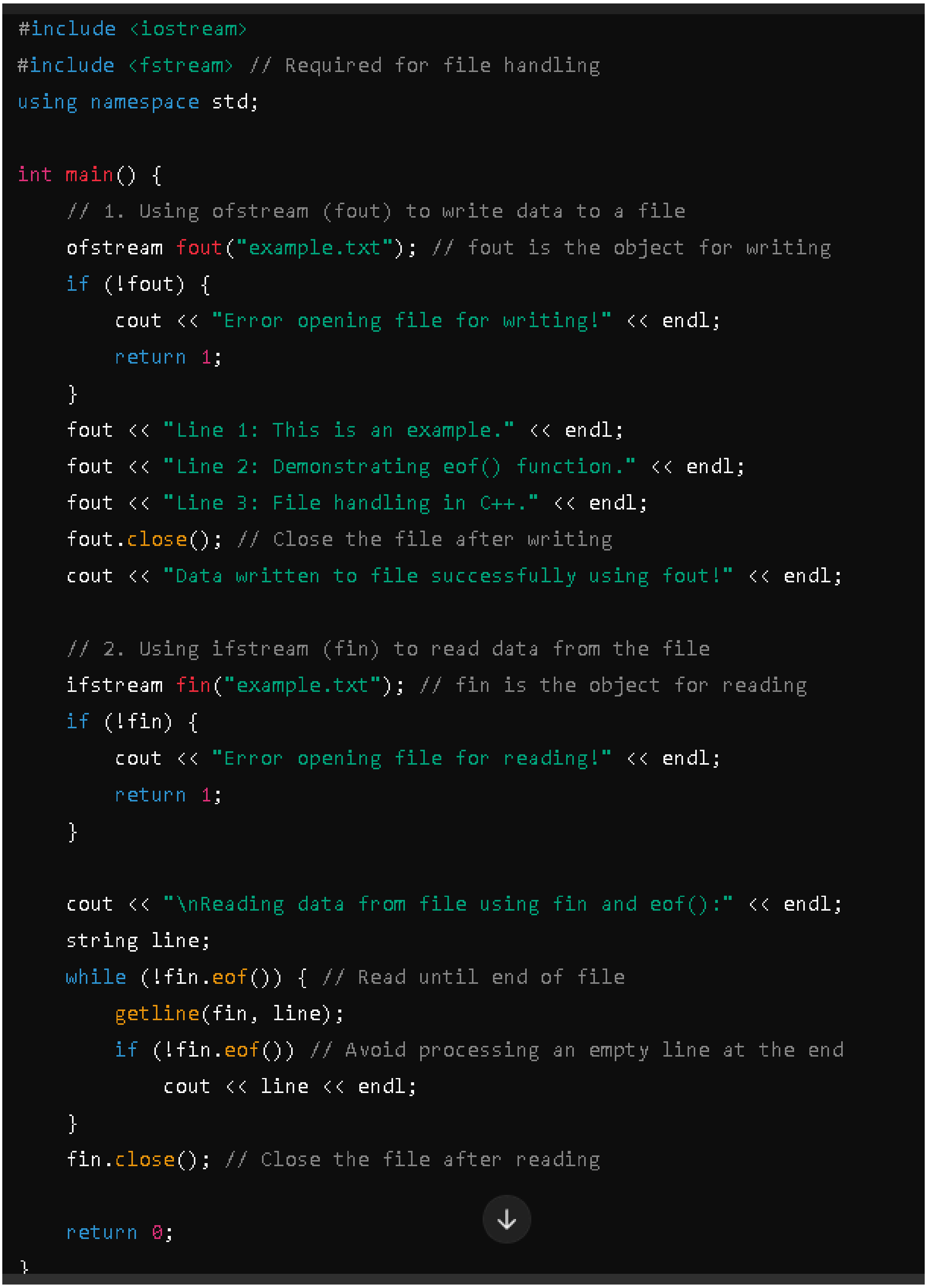


```
        cin.getline(name, 40);
        cout<<"Enter Roll No:";
        cin>>no;
        cout<<"Enter Marks:";
        cin>>marks;

        fout<<"Roll No: "<<no<<endl;
        fout<<"Name : "<< name <<endl;
        fout<<"Marks : "<<marks;
        fout.close();
        cout<<"Data written to file...";
    }
    //else
    //{
    //   cout<<"Enter file name at command line";
    //}
    return 0;
}
```

3) What are fstream, ifstream and ofstream? Illustrate with help of example.

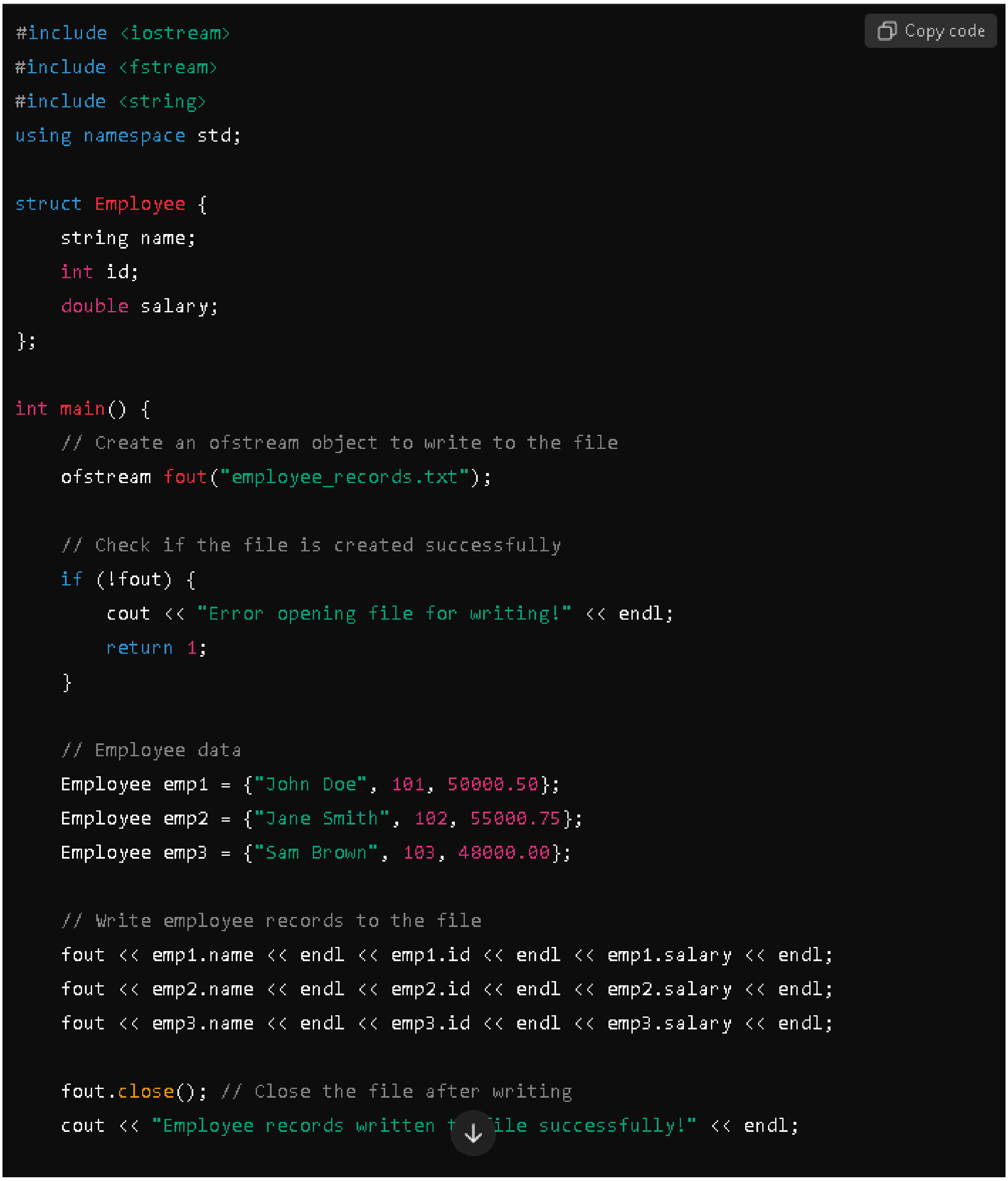


2.	Ifstream	Provides input operations and Contains open( ) with default input mode. Inherits the functions get( ), getline(), read( ), seekg( ), and tellg( ) from istream.
3.	ofstream	Provides output operations, contains open( ) with default output mode. Inherits put( ), seekp( ), tellp( ) and writer( ) functions from ostream.
4.	fstream	Provides support for simultaneous input and output operations. Contains open( ) with default input mode. Inherits all the functions from istream and ostream classes through iostream.

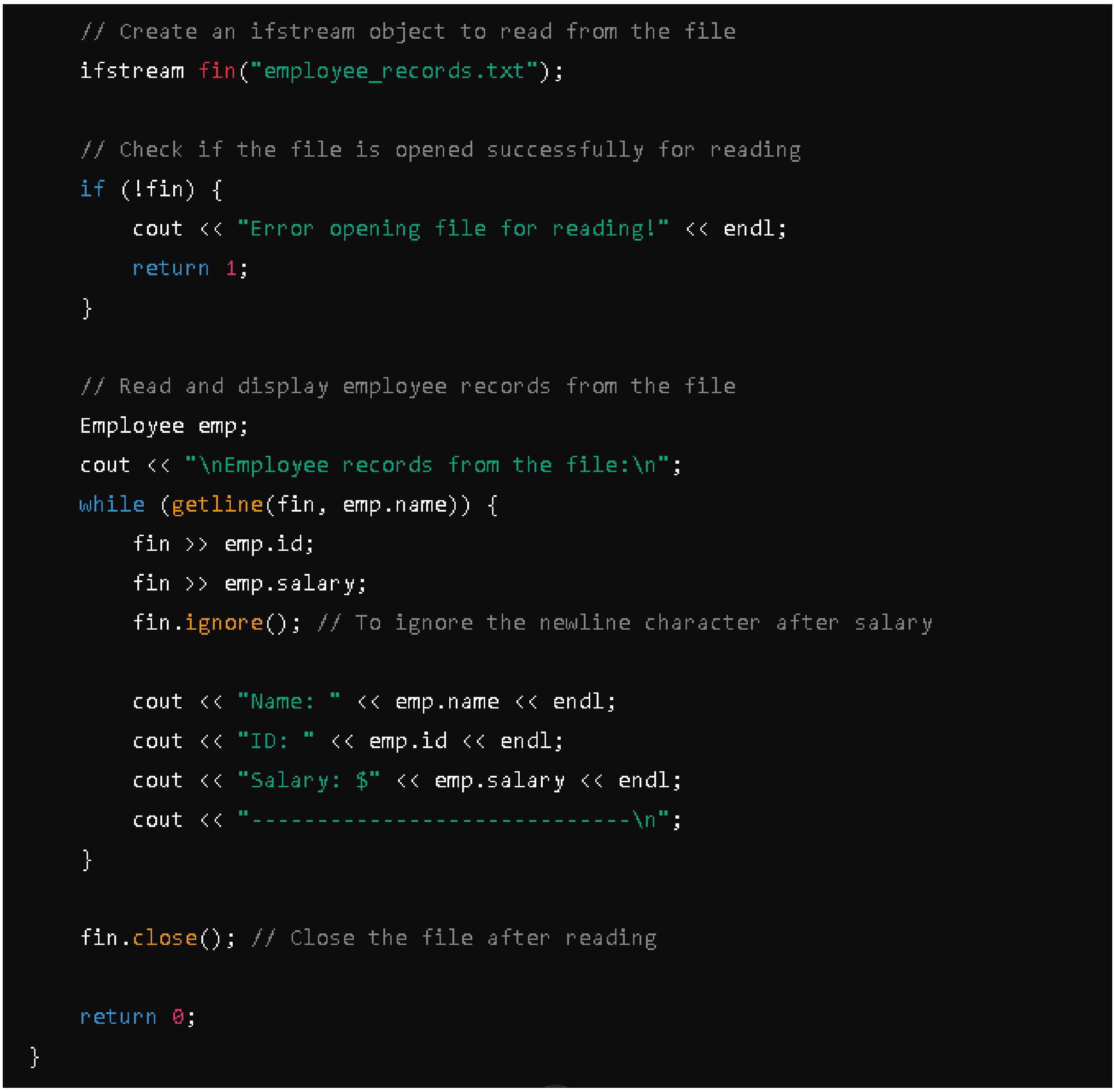


```
#include <iostream>
#include <fstream> // Required for file handling
using namespace std;

int main() {
    // 1. Using ofstream (fout) to write data to a file
    ofstream fout("example.txt"); // fout is the object for writing
    if (!fout) {
        cout << "Error opening file for writing!" << endl;
        return 1;
    }
    fout << "Line 1: This is an example." << endl;
    fout << "Line 2: Demonstrating eof() function." << endl;
    fout << "Line 3: File handling in C++." << endl;
    fout.close(); // Close the file after writing
    cout << "Data written to file successfully using fout!" << endl;

    // 2. Using ifstream (fin) to read data from the file
    ifstream fin("example.txt"); // fin is the object for reading
    if (!fin) {
        cout << "Error opening file for reading!" << endl;
        return 1;
    }

    cout << "\nReading data from file using fin and eof():" << endl;
    string line;
    while (!fin.eof()) { // Read until end of file
        getline(fin, line);
        if (!fin.eof()) // Avoid processing an empty line at the end
            cout << line << endl;
    }
    fin.close(); // Close the file after reading

    return 0;
}
```

4) **Write a program to create file, read and write record into it. Every record contains employee name, id and salary. Store and retrieve atleast 3 data**



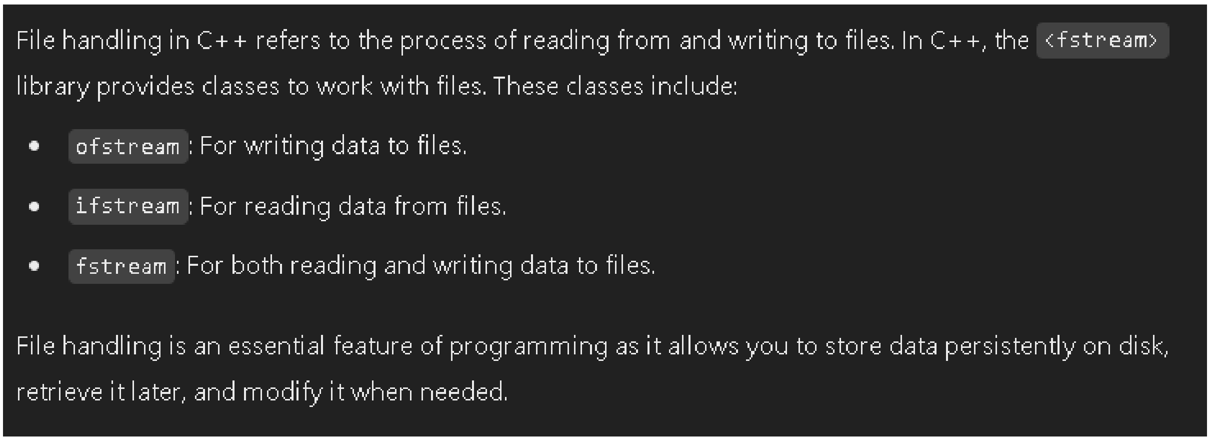
```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

struct Employee {
    string name;
    int id;
    double salary;
};

int main() {
    // Create an ofstream object to write to the file
    ofstream fout("employee_records.txt");

    // Check if the file is created successfully
    if (!fout) {
        cout << "Error opening file for writing!" << endl;
        return 1;
    }

    // Employee data
    Employee emp1 = {"John Doe", 101, 50000.50};
    Employee emp2 = {"Jane Smith", 102, 55000.75};
    Employee emp3 = {"Sam Brown", 103, 48000.00};

    // Write employee records to the file
    fout << emp1.name << endl << emp1.id << endl << emp1.salary << endl;
    fout << emp2.name << endl << emp2.id << endl << emp2.salary << endl;
    fout << emp3.name << endl << emp3.id << endl << emp3.salary << endl;

    fout.close(); // Close the file after writing
    cout << "Employee records written t   ile successfully!" << endl;
```



```cpp
    // Create an ifstream object to read from the file
    ifstream fin("employee_records.txt");

    // Check if the file is opened successfully for reading
    if (!fin) {
        cout << "Error opening file for reading!" << endl;
        return 1;
    }

    // Read and display employee records from the file
    Employee emp;
    cout << "\nEmployee records from the file:\n";
    while (getline(fin, emp.name)) {
        fin >> emp.id;
        fin >> emp.salary;
        fin.ignore(); // To ignore the newline character after salary

        cout << "Name: " << emp.name << endl;
        cout << "ID: " << emp.id << endl;
        cout << "Salary: $" << emp.salary << endl;
        cout << "----------------------------\n";
    }

    fin.close(); // Close the file after reading

    return 0;
}
```

5) What do you mean by file handling? Explain the following functions.

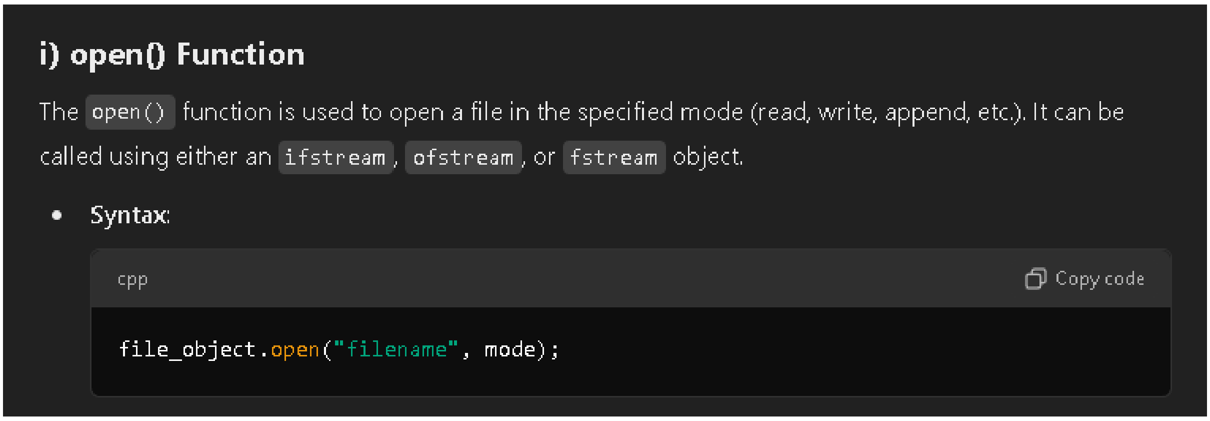
i) open() ii) get() iii) getline()



File handling in C++ refers to the process of reading from and writing to files. In C++, the `<fstream>` library provides classes to work with files. These classes include:

- `ofstream`: For writing data to files.
- `ifstream`: For reading data from files.
- `fstream`: For both reading and writing data to files.

File handling is an essential feature of programming as it allows you to store data persistently on disk, retrieve it later, and modify it when needed.

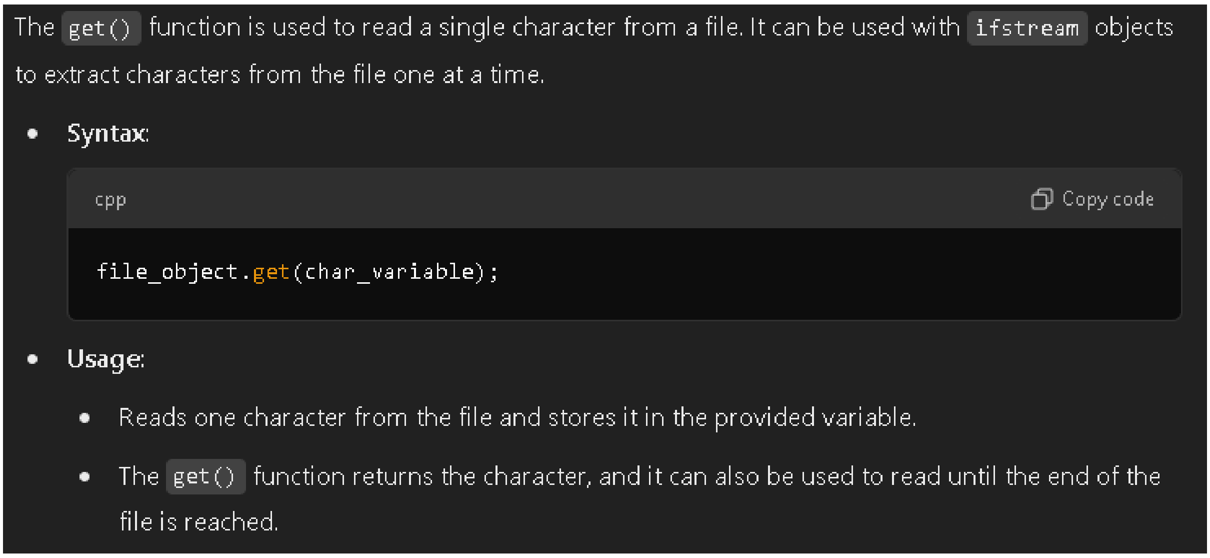


i) open() Function

The `open()` function is used to open a file in the specified mode (read, write, append, etc.). It can be called using either an `ifstream`, `ofstream`, or `fstream` object.

- Syntax:

```cpp
file_object.open("filename", mode);
```

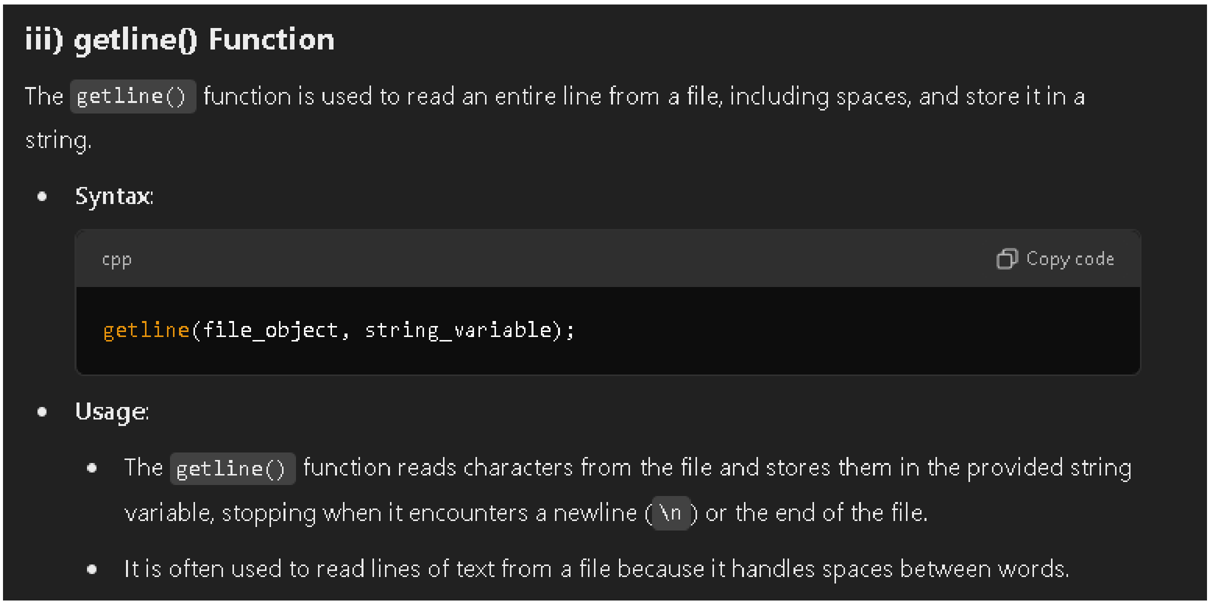


The `get()` function is used to read a single character from a file. It can be used with `ifstream` objects to extract characters from the file one at a time.

- Syntax:

```cpp
file_object.get(char_variable);
```

- Usage:
  - Reads one character from the file and stores it in the provided variable.
  - The `get()` function returns the character, and it can also be used to read until the end of the file is reached.



iii) getline() Function

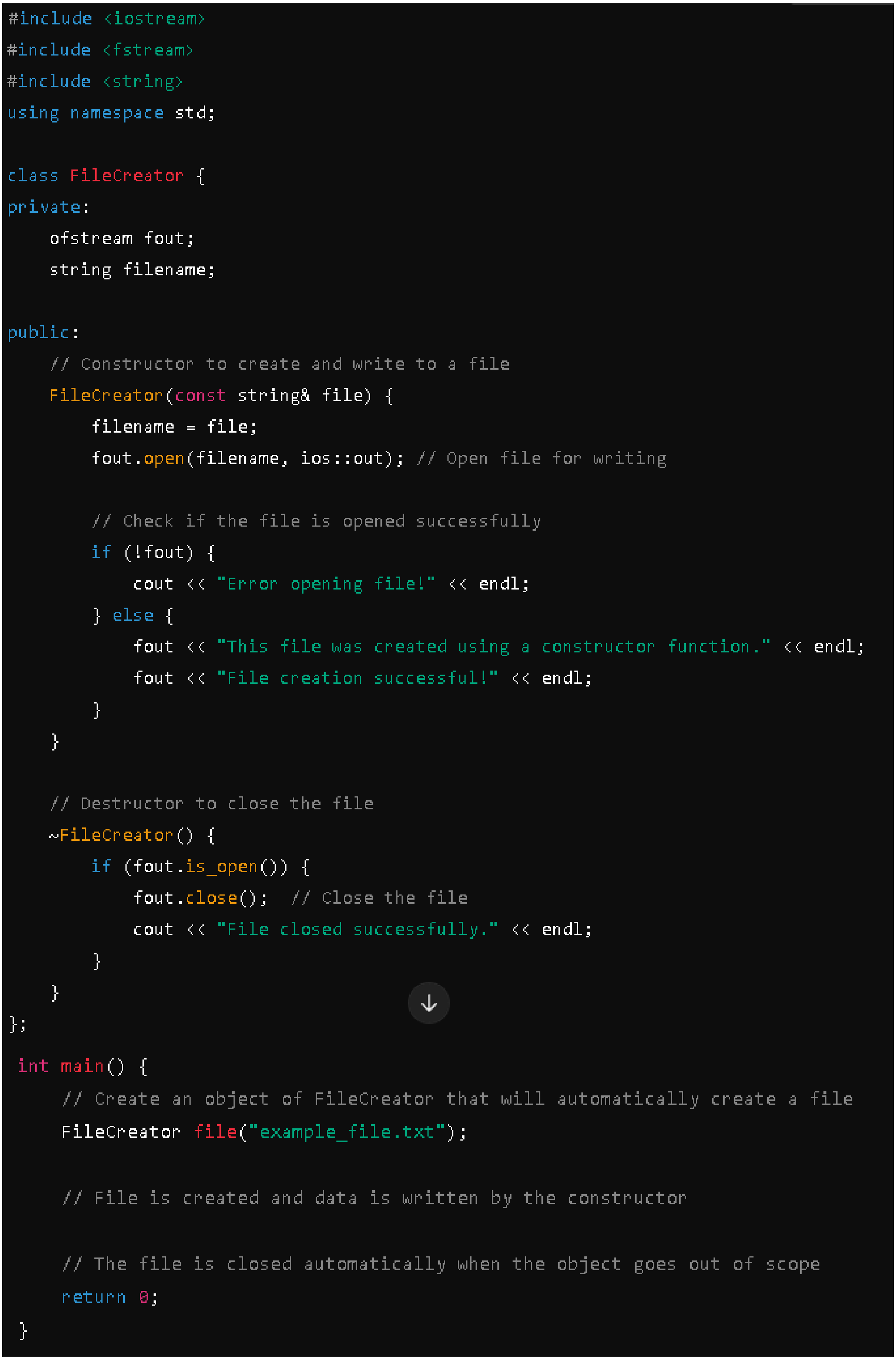
The `getline()` function is used to read an entire line from a file, including spaces, and store it in a string.

- **Syntax:**

```cpp
getline(file_object, string_variable);
```

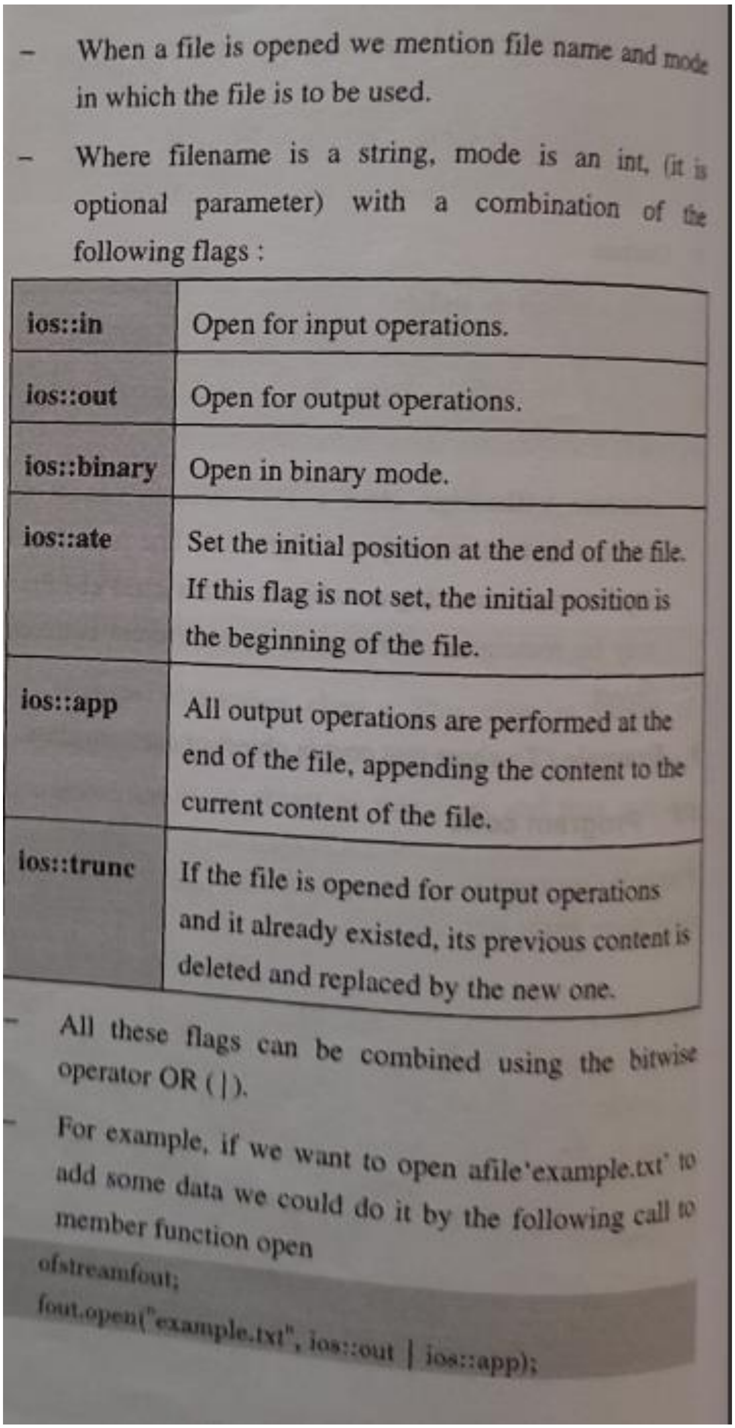
- **Usage:**
  - The `getline()` function reads characters from the file and stores them in the provided string variable, stopping when it encounters a newline (`\n`) or the end of the file.
  - It is often used to read lines of text from a file because it handles spaces between words.

6) **Write a program to create files using constructor function.**



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

class FileCreator {
private:
    ofstream fout;
    string filename;

public:
    // Constructor to create and write to a file
    FileCreator(const string& file) {
        filename = file;
        fout.open(filename, ios::out); // Open file for writing

        // Check if the file is opened successfully
        if (!fout) {
            cout << "Error opening file!" << endl;
        } else {
            fout << "This file was created using a constructor function." << endl;
            fout << "File creation successful!" << endl;
        }
    }

    // Destructor to close the file
    ~FileCreator() {
        if (fout.is_open()) {
            fout.close();  // Close the file
            cout << "File closed successfully." << endl;
        }
    }
};
 int main() {
     // Create an object of FileCreator that will automatically create a file
     FileCreator file("example_file.txt");

     // File is created and data is written by the constructor

     // The file is closed automatically when the object goes out of scope
     return 0;
 }
```

7) What are different file opening mode?

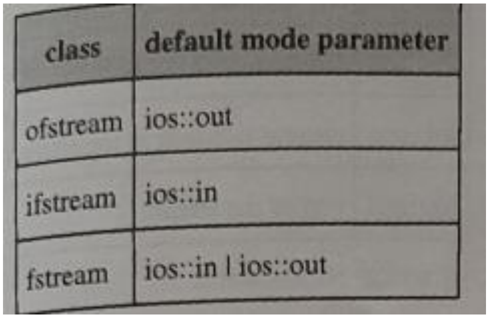


- When a file is opened we mention file name and mode in which the file is to be used.
- Where filename is a string, mode is an int, (it is optional parameter) with a combination of the following flags :

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

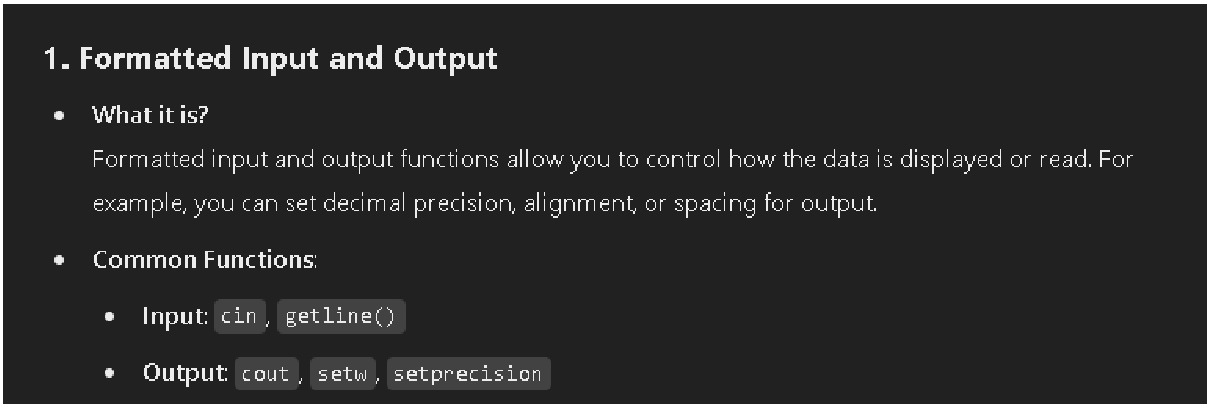
- All these flags can be combined using the bitwise operator OR (|).
- For example, if we want to open afile'example.txt' to add some data we could do it by the following call to member function open

```
ofstreamfout;
fout.open("example.txt", ios::out | ios::app);
```



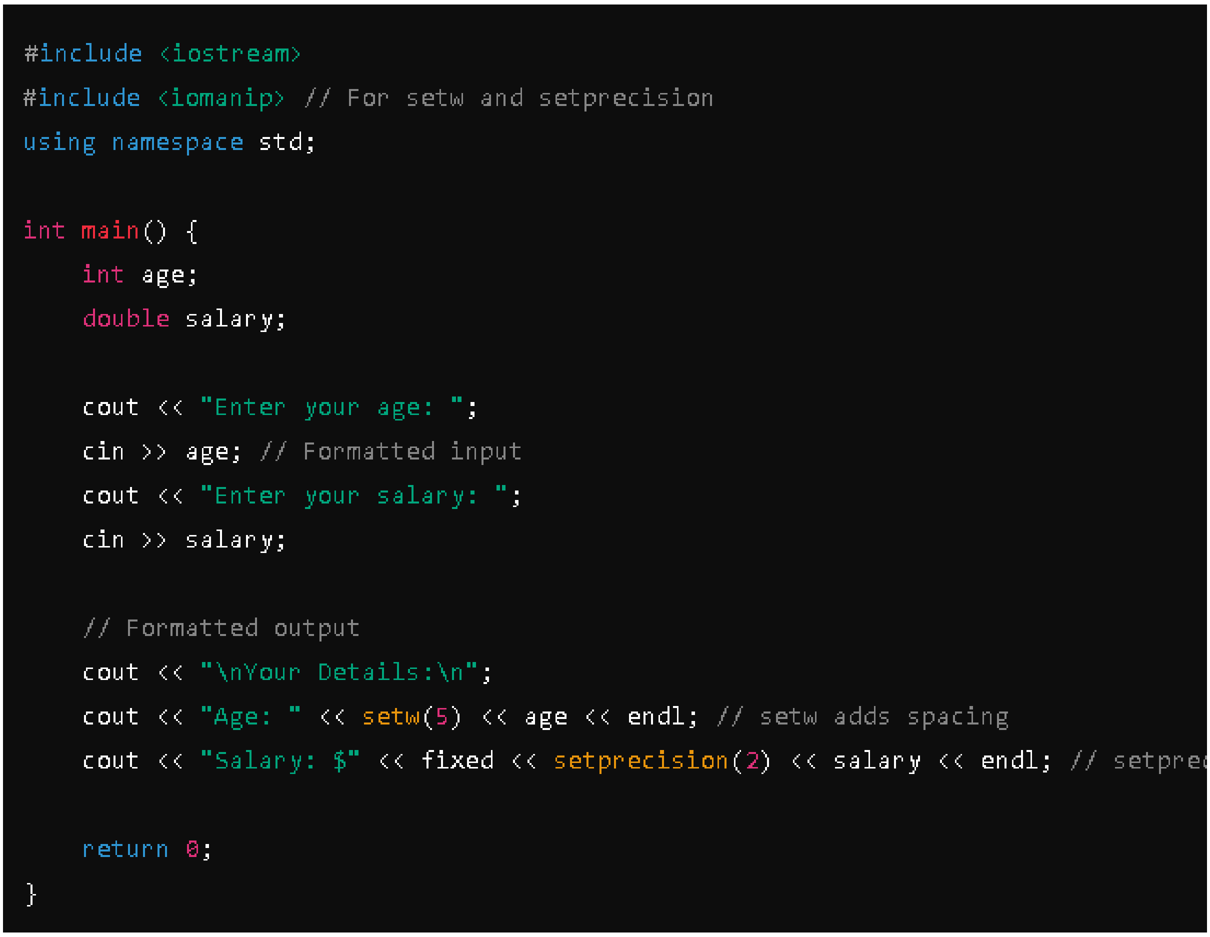
class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

8) Explain formatted and unformatted input and output functions used in C++ with example.

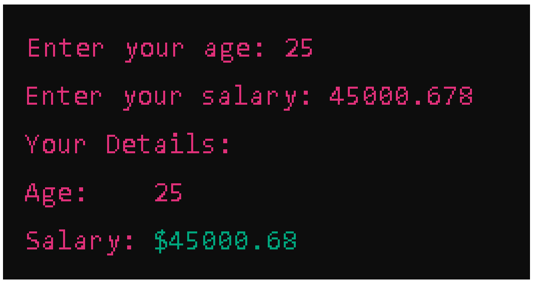


1. Formatted Input and Output

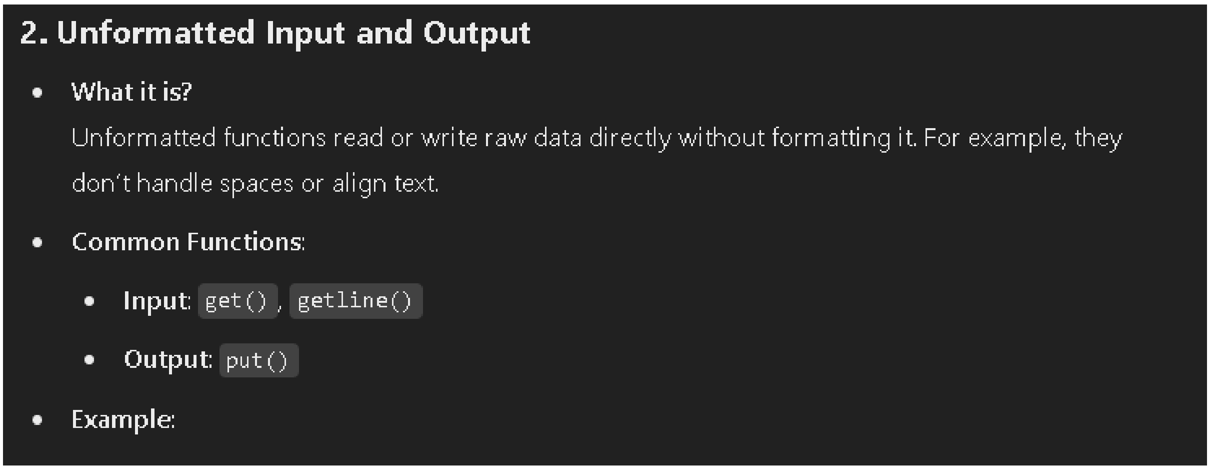
- **What it is?**
  Formatted input and output functions allow you to control how the data is displayed or read. For example, you can set decimal precision, alignment, or spacing for output.
- **Common Functions:**
  - **Input:** `cin`, `getline()`
  - **Output:** `cout`, `setw`, `setprecision`



```cpp
#include <iostream>
#include <iomanip> // For setw and setprecision
using namespace std;

int main() {
    int age;
    double salary;

    cout << "Enter your age: ";
    cin >> age; // Formatted input
    cout << "Enter your salary: ";
    cin >> salary;

    // Formatted output
    cout << "\nYour Details:\n";
    cout << "Age: " << setw(5) << age << endl; // setw adds spacing
    cout << "Salary: $" << fixed << setprecision(2) << salary << endl; // setprec

    return 0;
}
```



```
Enter your age: 25
Enter your salary: 45000.678
Your Details:
Age:    25
Salary: $45000.68
```

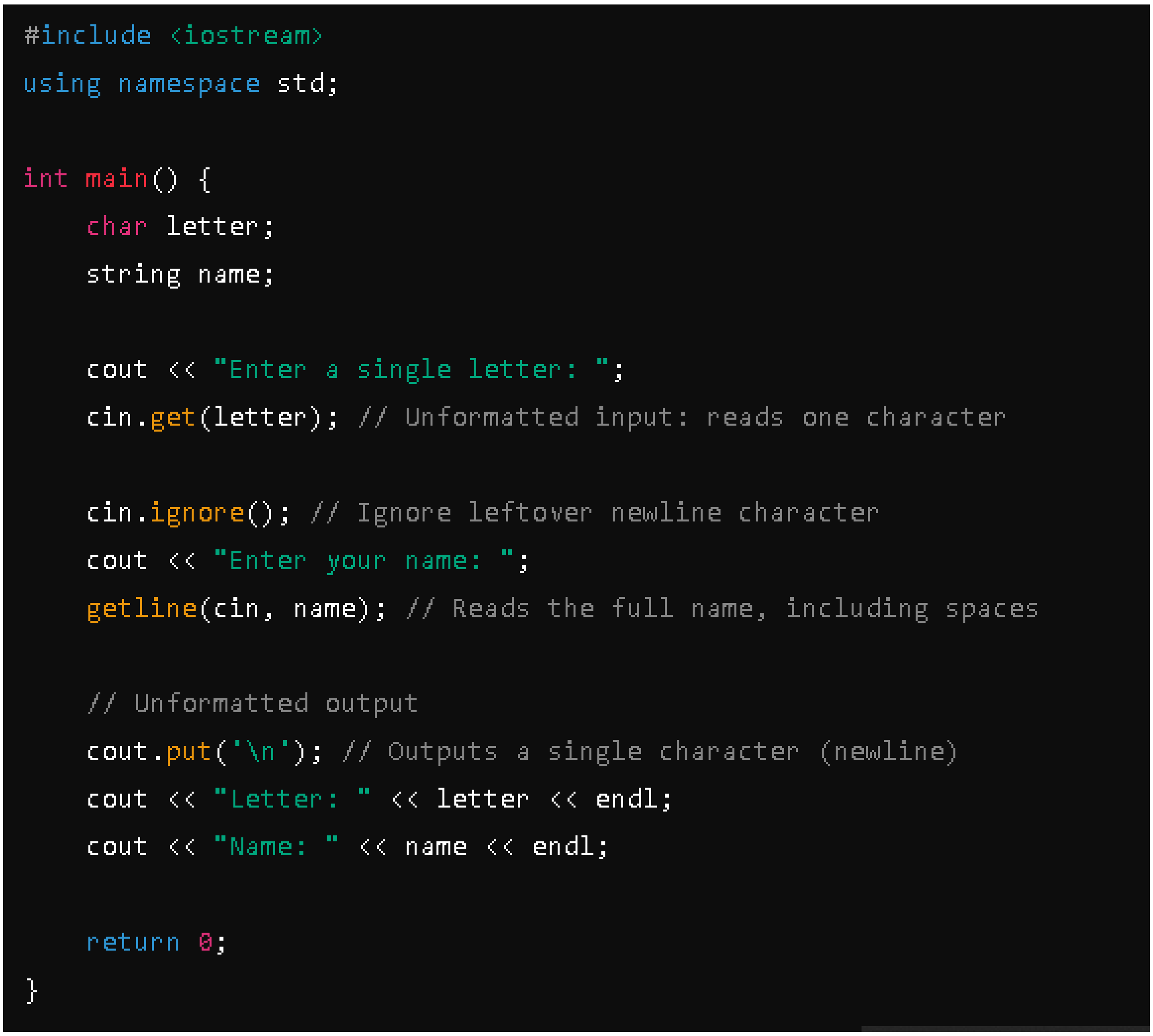


2. Unformatted Input and Output

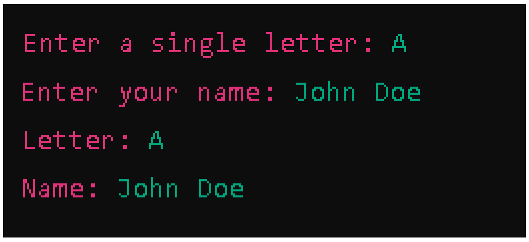
- **What it is?**

  Unformatted functions read or write raw data directly without formatting it. For example, they don't handle spaces or align text.

- **Common Functions:**
  - **Input:** `get()`, `getline()`
  - **Output:** `put()`
- **Example:**

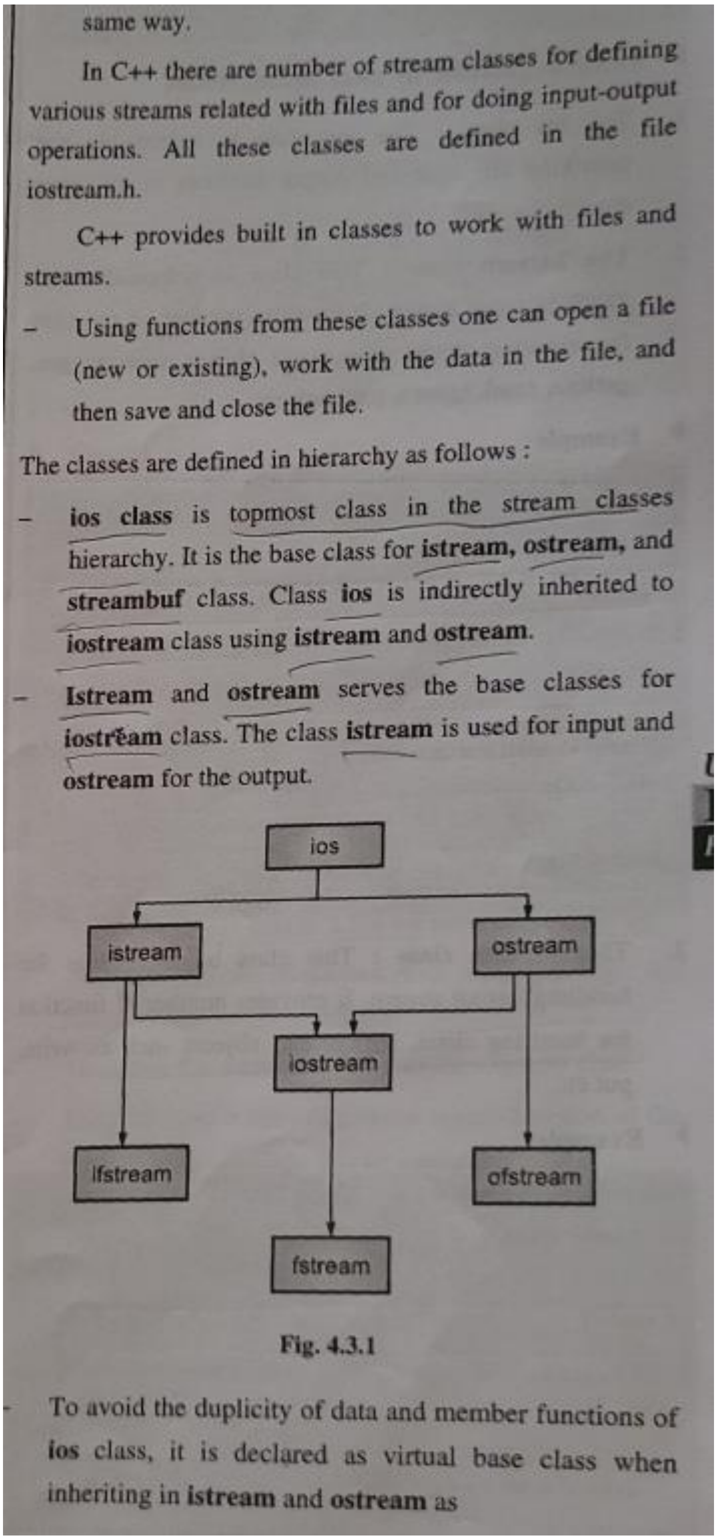


```cpp
#include <iostream>
using namespace std;

int main() {
    char letter;
    string name;

    cout << "Enter a single letter: ";
    cin.get(letter); // Unformatted input: reads one character

    cin.ignore(); // Ignore leftover newline character
    cout << "Enter your name: ";
    getline(cin, name); // Reads the full name, including spaces

    // Unformatted output
    cout.put('\n'); // Outputs a single character (newline)
    cout << "Letter: " << letter << endl;
    cout << "Name: " << name << endl;

    return 0;
}
```



```
Enter a single letter: A
Enter your name: John Doe
Letter: A
Name: John Doe
```

9) What are stream classes and their use? Provide the hierarchy of stream classes in C++



same way.

In C++ there are number of stream classes for defining various streams related with files and for doing input-output operations. All these classes are defined in the file iostream.h.

C++ provides built in classes to work with files and streams.

- Using functions from these classes one can open a file (new or existing), work with the data in the file, and then save and close the file.

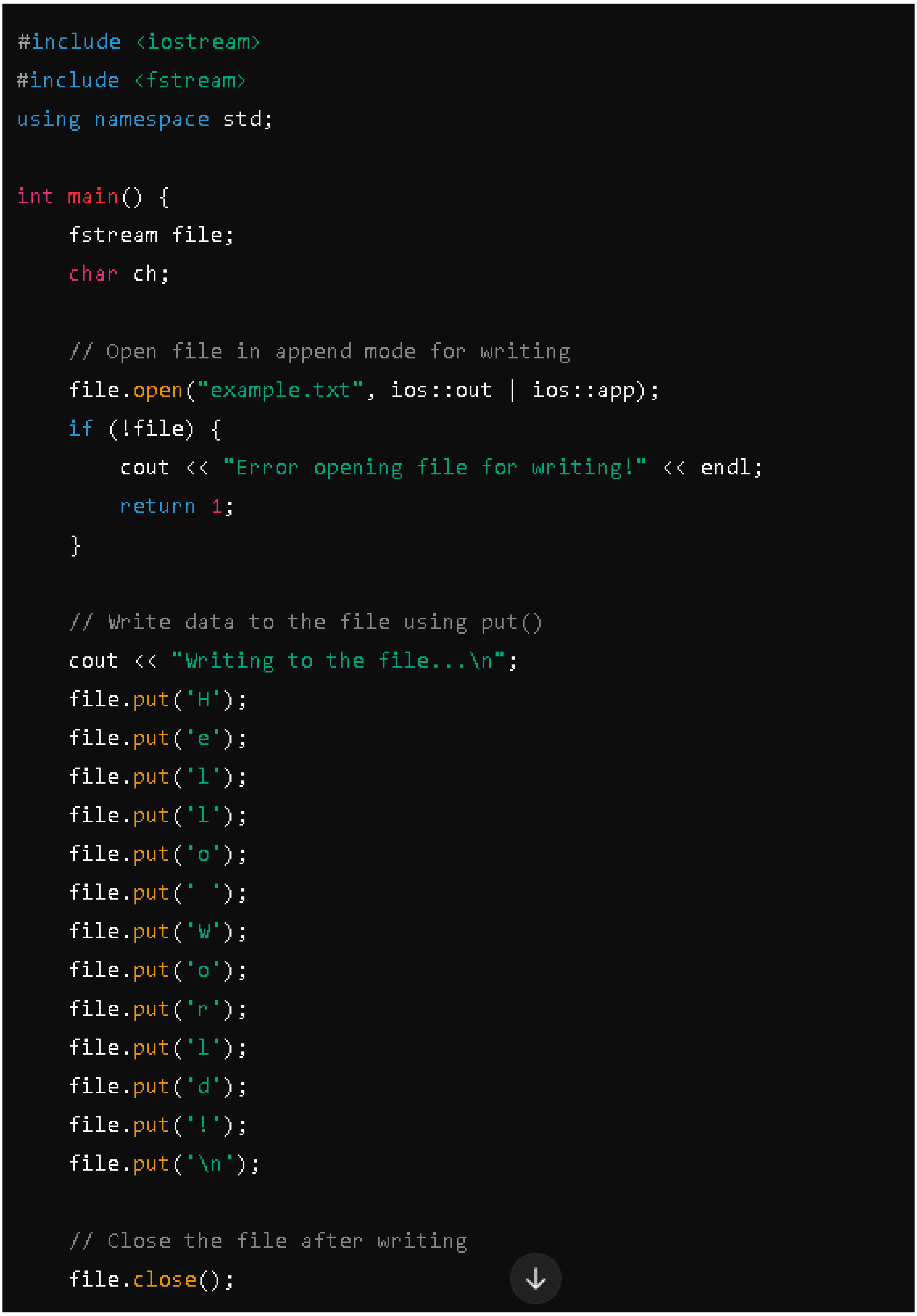
The classes are defined in hierarchy as follows :

- **ios class** is topmost class in the stream classes hierarchy. It is the base class for **istream, ostream,** and **streambuf** class. Class **ios** is indirectly inherited to **iostream** class using **istream** and **ostream**.
- **Istream** and **ostream** serves the base classes for **iostream** class. The class **istream** is used for input and **ostream** for the output.

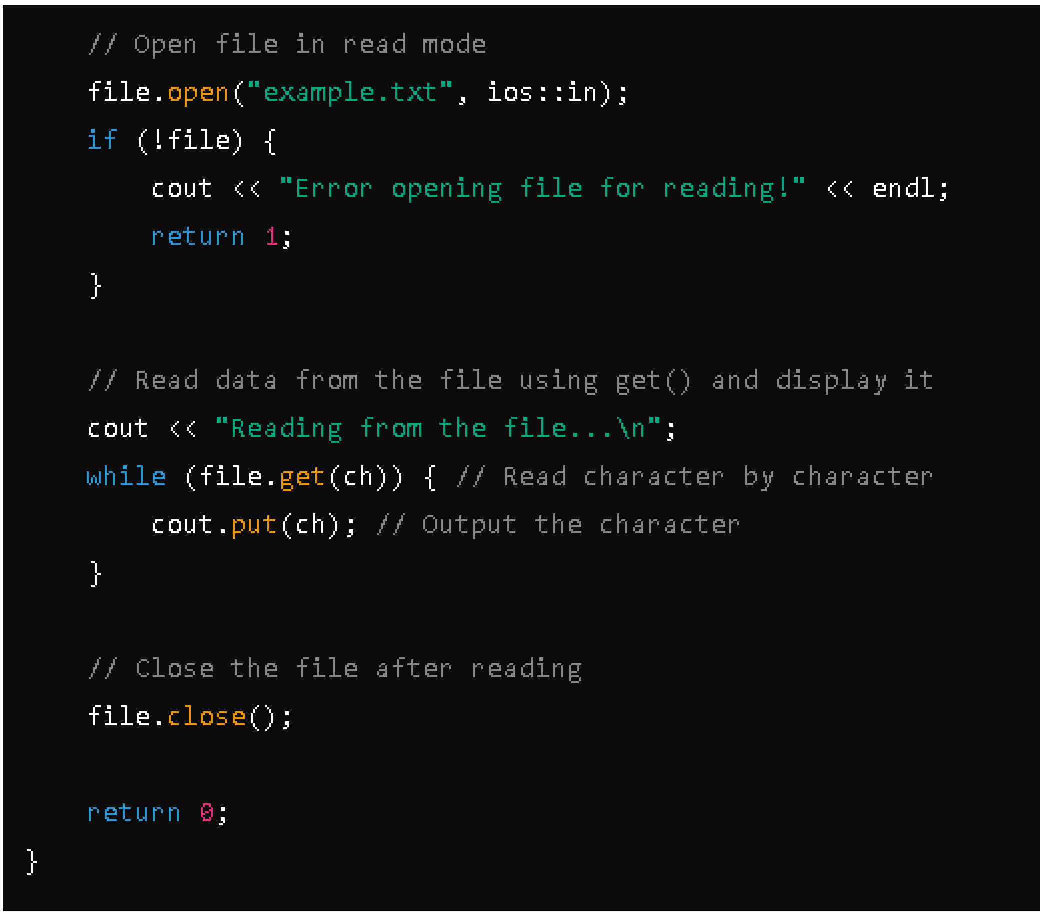
Fig. 4.3.1

- To avoid the duplicity of data and member functions of **ios** class, it is declared as virtual base class when inheriting in **istream** and **ostream** as

10) **Write a program Using the C++ file input and output class with open(), get(), put(),close() methods for opening, reading from and writing to a file. Use append mode while opening the file for writing.**

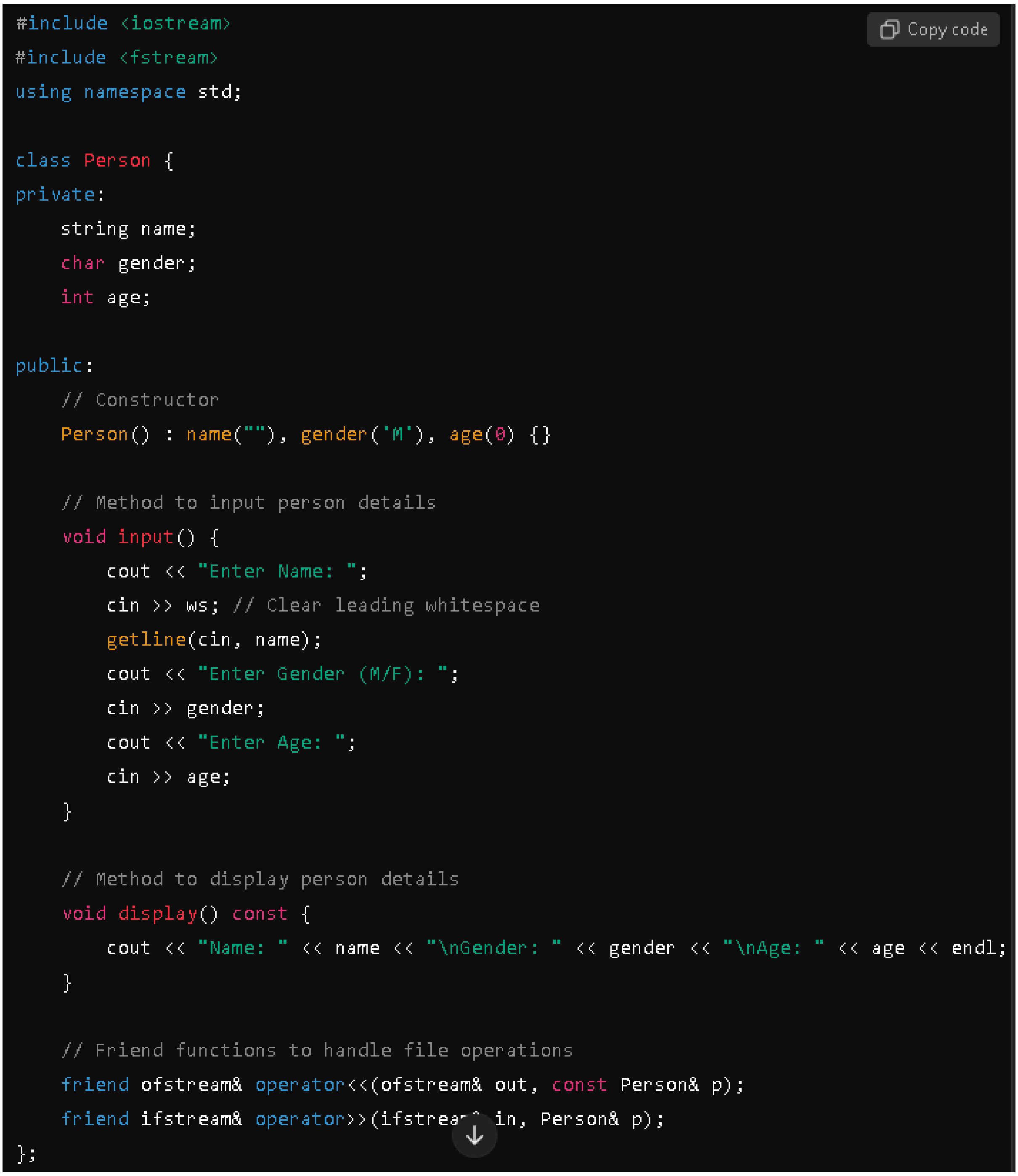


```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream file;
    char ch;

    // Open file in append mode for writing
    file.open("example.txt", ios::out | ios::app);
    if (!file) {
        cout << "Error opening file for writing!" << endl;
        return 1;
    }

    // Write data to the file using put()
    cout << "Writing to the file...\n";
    file.put('H');
    file.put('e');
    file.put('l');
    file.put('l');
    file.put('o');
    file.put(' ');
    file.put('W');
    file.put('o');
    file.put('r');
    file.put('l');
    file.put('d');
    file.put('!');
    file.put('\n');

    // Close the file after writing
    file.close();
```

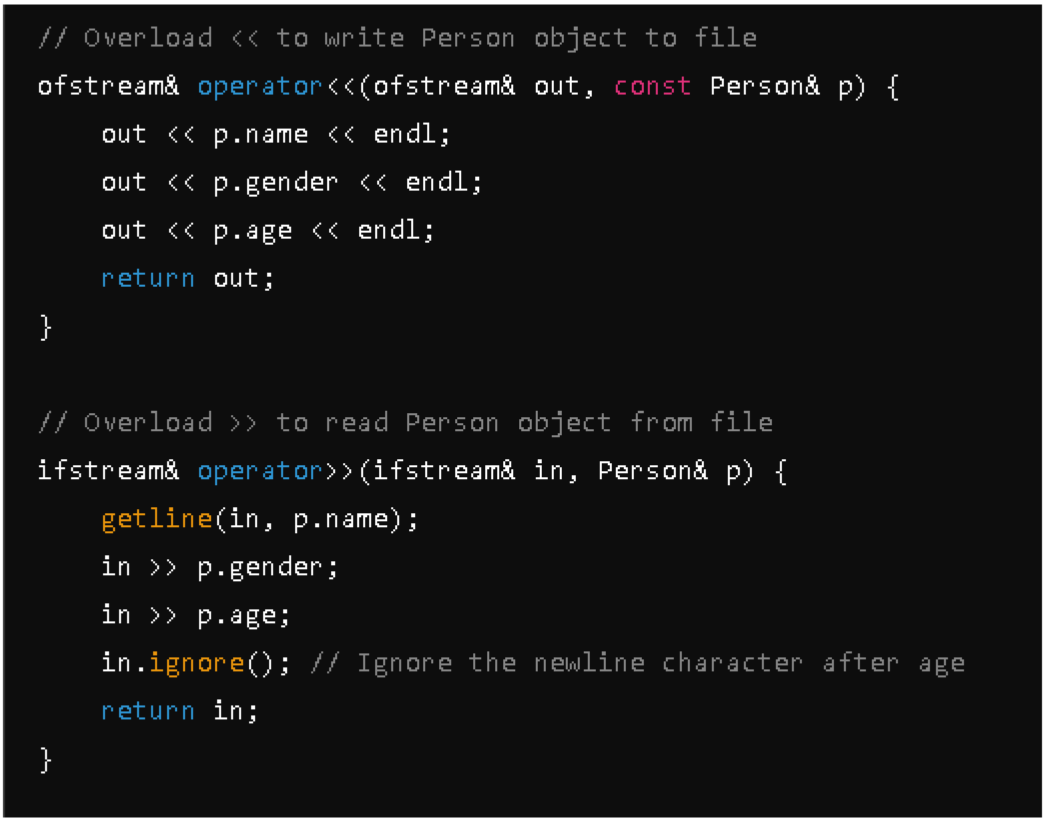


```
    // Open file in read mode
    file.open("example.txt", ios::in);
    if (!file) {
        cout << "Error opening file for reading!" << endl;
        return 1;
    }

    // Read data from the file using get() and display it
    cout << "Reading from the file...\n";
    while (file.get(ch)) { // Read character by character
        cout.put(ch); // Output the character
    }

    // Close the file after reading
    file.close();

    return 0;
}
```

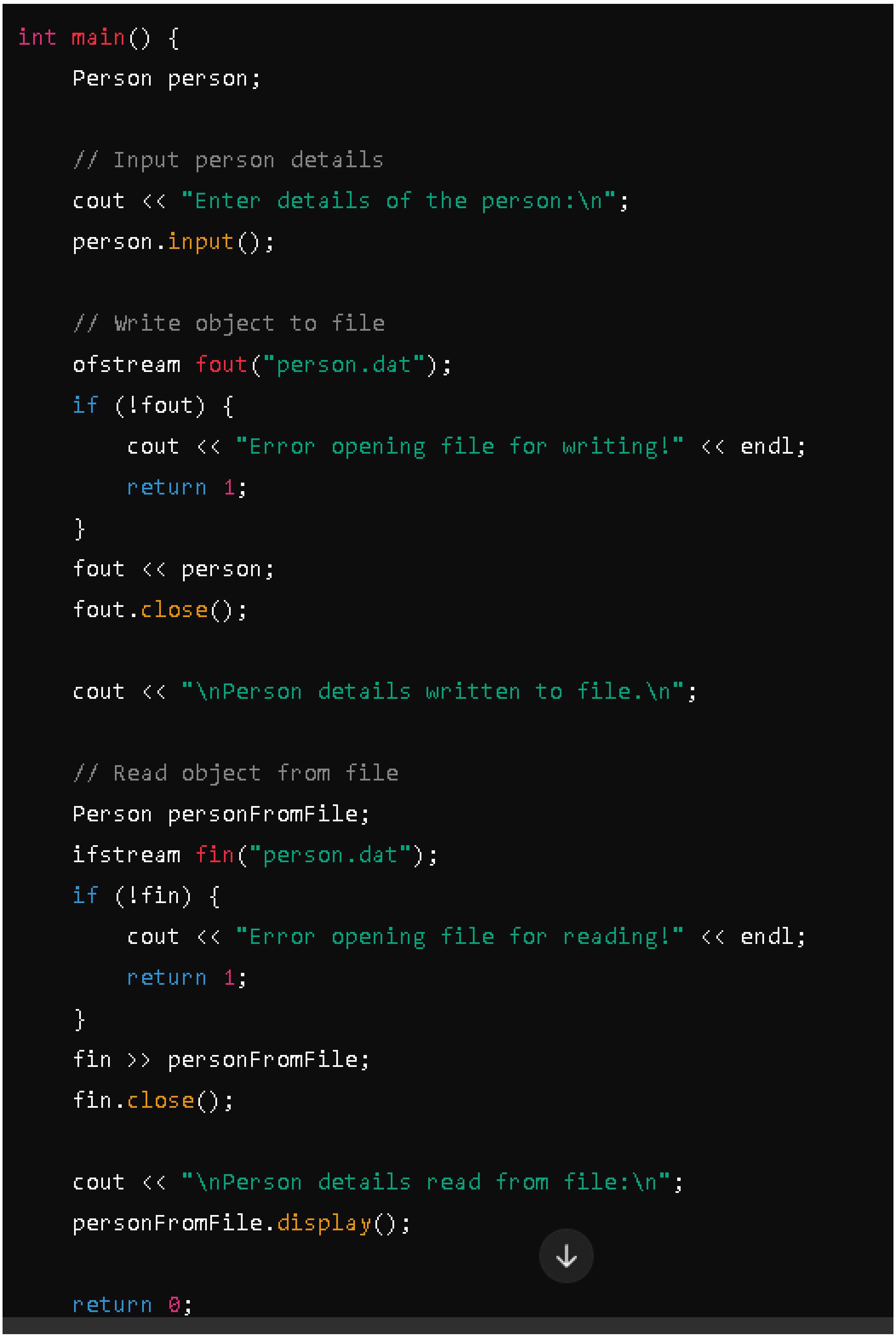
11) **Define a class Person that has three attributes viz name, gender and age. Write a C++ Program that writes an object to a file and reads an object from a file.**



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

class Person {
private:
    string name;
    char gender;
    int age;

public:
    // Constructor
    Person() : name(""), gender('M'), age(0) {}

    // Method to input person details
    void input() {
        cout << "Enter Name: ";
        cin >> ws; // Clear leading whitespace
        getline(cin, name);
        cout << "Enter Gender (M/F): ";
        cin >> gender;
        cout << "Enter Age: ";
        cin >> age;
    }

    // Method to display person details
    void display() const {
        cout << "Name: " << name << "\nGender: " << gender << "\nAge: " << age << endl;
    }

    // Friend functions to handle file operations
    friend ofstream& operator<<(ofstream& out, const Person& p);
    friend ifstream& operator>>(ifstream& in, Person& p);
};
```

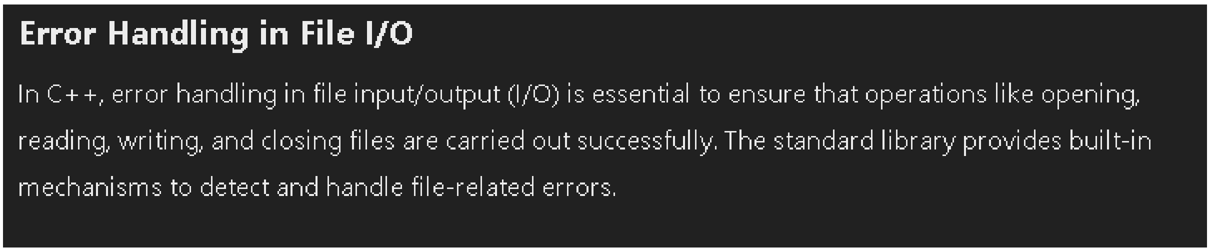


```
// Overload << to write Person object to file
ofstream& operator<<(ofstream& out, const Person& p) {
    out << p.name << endl;
    out << p.gender << endl;
    out << p.age << endl;
    return out;
}

// Overload >> to read Person object from file
ifstream& operator>>(ifstream& in, Person& p) {
    getline(in, p.name);
    in >> p.gender;
    in >> p.age;
    in.ignore(); // Ignore the newline character after age
    return in;
}
```



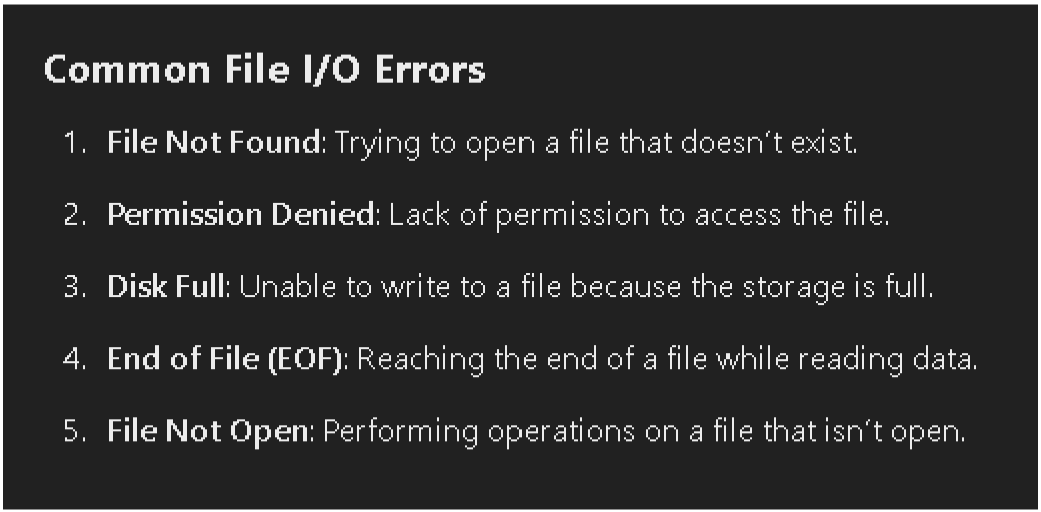
```
int main() {
    Person person;

    // Input person details
    cout << "Enter details of the person:\n";
    person.input();

    // Write object to file
    ofstream fout("person.dat");
    if (!fout) {
        cout << "Error opening file for writing!" << endl;
        return 1;
    }
    fout << person;
    fout.close();

    cout << "\nPerson details written to file.\n";

    // Read object from file
    Person personFromFile;
    ifstream fin("person.dat");
    if (!fin) {
        cout << "Error opening file for reading!" << endl;
        return 1;
    }
    fin >> personFromFile;
    fin.close();

    cout << "\nPerson details read from file:\n";
    personFromFile.display();

    return 0;
```

12) Explain the errror handling in file I/O



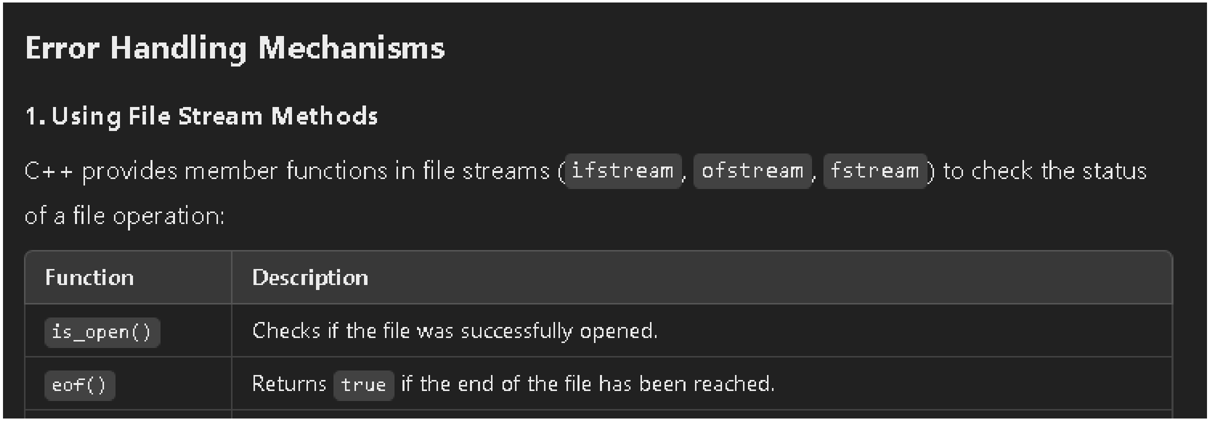
Error Handling in File I/O

In C++, error handling in file input/output (I/O) is essential to ensure that operations like opening, reading, writing, and closing files are carried out successfully. The standard library provides built-in mechanisms to detect and handle file-related errors.



Common File I/O Errors

1. **File Not Found**: Trying to open a file that doesn't exist.
2. **Permission Denied**: Lack of permission to access the file.
3. **Disk Full**: Unable to write to a file because the storage is full.
4. **End of File (EOF)**: Reaching the end of a file while reading data.
5. **File Not Open**: Performing operations on a file that isn't open.

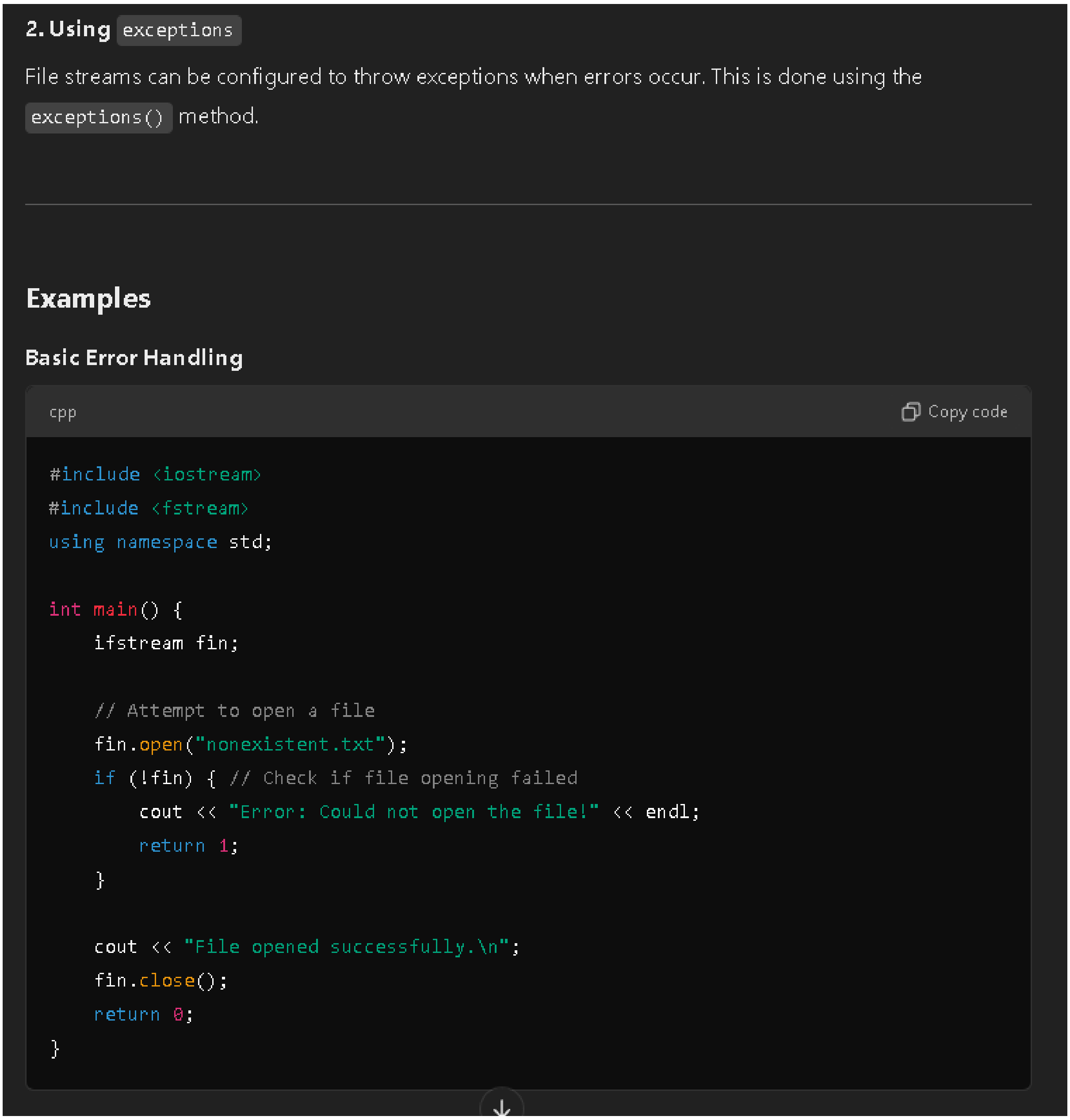


Error Handling Mechanisms

1. Using File Stream Methods

C++ provides member functions in file streams (`ifstream`, `ofstream`, `fstream`) to check the status of a file operation:

Function	Description
`is_open()`	Checks if the file was successfully opened.
`eof()`	Returns `true` if the end of the file has been reached.



2. Using `exceptions`

File streams can be configured to throw exceptions when errors occur. This is done using the `exceptions()` method.

Examples

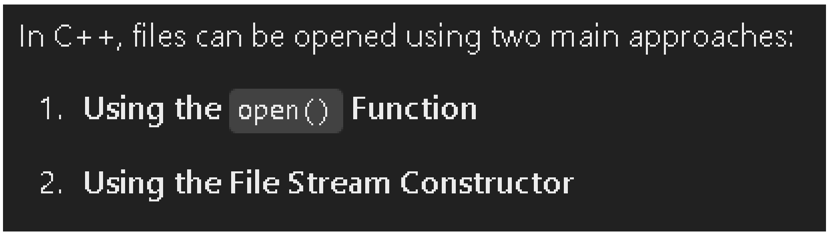
Basic Error Handling

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

int main() {
    ifstream fin;

    // Attempt to open a file
    fin.open("nonexistent.txt");
    if (!fin) { // Check if file opening failed
        cout << "Error: Could not open the file!" << endl;
        return 1;
    }

    cout << "File opened successfully.\n";
    fin.close();
    return 0;
}
```

Checking End of File (EOF)

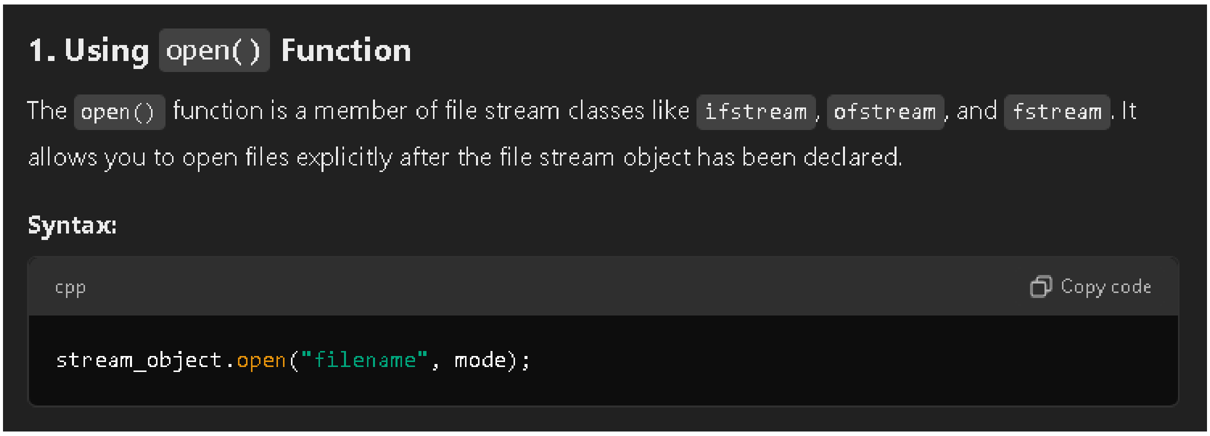
```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ifstream fin("example.txt");
    if (!fin) {
        cout << "Error: Could not open the file!" << endl;
        return 1;
    }

    char ch;
    while (!fin.eof()) { // Loop until end of file
        fin.get(ch);
        if (fin) { // Check if reading was successful
            cout << ch;
        }
    }

    fin.close();
    return 0;
}
```

13) Explain the two ways in which files can be opened, open () and Using constructor with a program.



In C++, files can be opened using two main approaches:

1. **Using the `open()` Function**
2. **Using the File Stream Constructor**

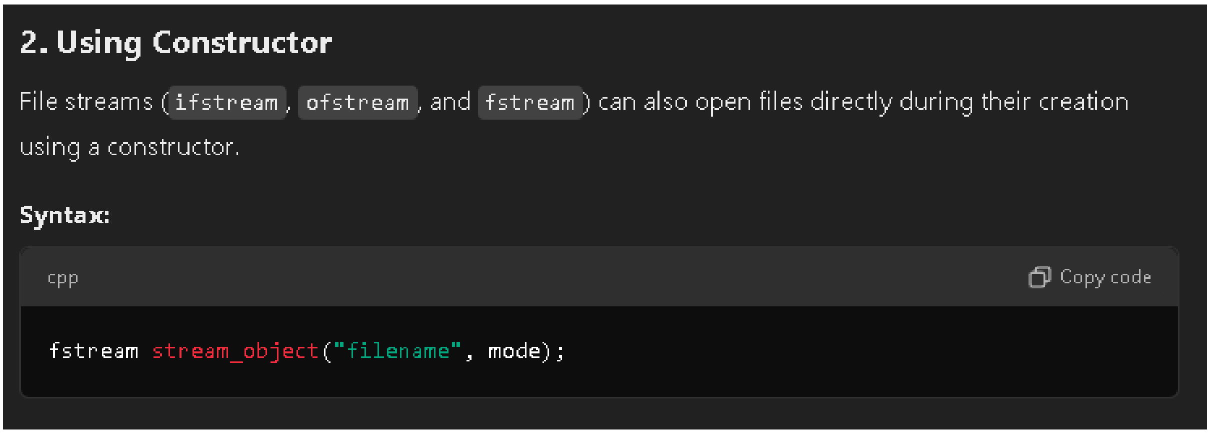


1. Using `open()` Function

The `open()` function is a member of file stream classes like `ifstream`, `ofstream`, and `fstream`. It allows you to open files explicitly after the file stream object has been declared.

Syntax:

```cpp
stream_object.open("filename", mode);
```

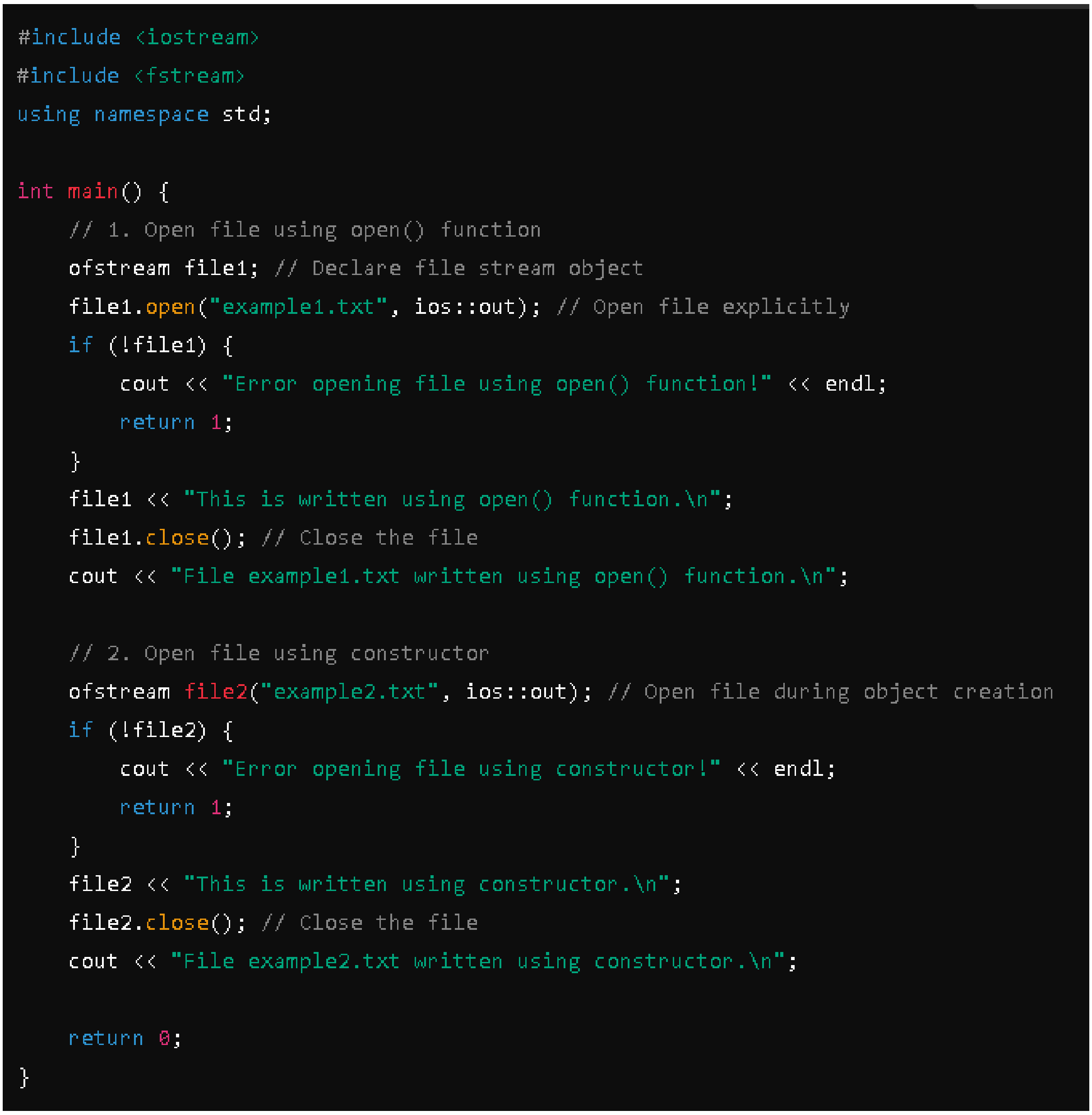


2. Using Constructor

File streams (`ifstream`, `ofstream`, and `fstream`) can also open files directly during their creation using a constructor.

Syntax:

```cpp
fstream stream_object("filename", mode);
```

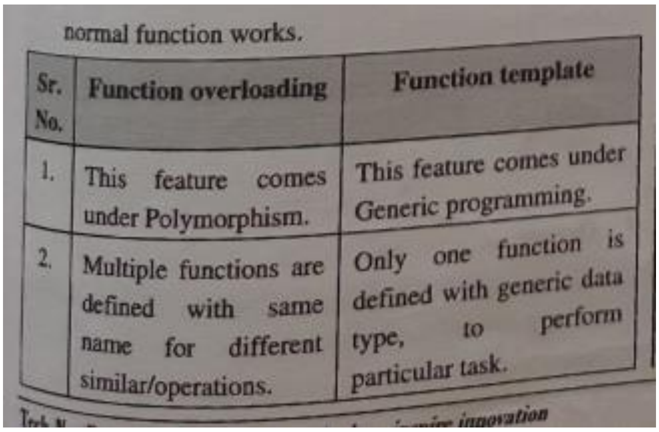


```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    // 1. Open file using open() function
    ofstream file1; // Declare file stream object
    file1.open("example1.txt", ios::out); // Open file explicitly
    if (!file1) {
        cout << "Error opening file using open() function!" << endl;
        return 1;
    }
    file1 << "This is written using open() function.\n";
    file1.close(); // Close the file
    cout << "File example1.txt written using open() function.\n";

    // 2. Open file using constructor
    ofstream file2("example2.txt", ios::out); // Open file during object creation
    if (!file2) {
        cout << "Error opening file using constructor!" << endl;
        return 1;
    }
    file2 << "This is written using constructor.\n";
    file2.close(); // Close the file
    cout << "File example2.txt written using constructor.\n";

    return 0;
}
```

UNIT 5

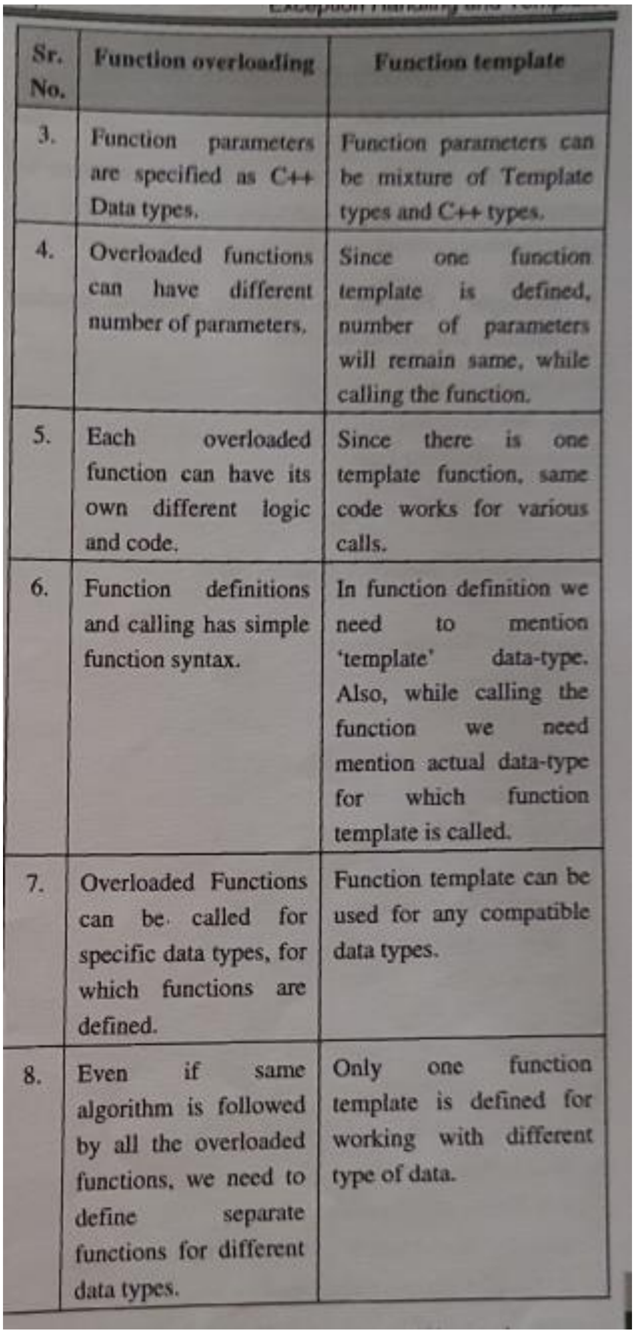
Exception Handling and Templates

1) Distinguish between overloaded function and function template with suitable example.



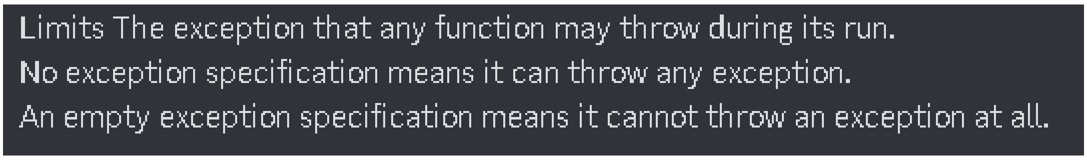
normal function works.

Sr. No.	Function overloading	Function template
1.	This feature comes under Polymorphism.	This feature comes under Generic programming.
2.	Multiple functions are defined with same name for different similar/operations.	Only one function is defined with generic data type, to perform particular task.

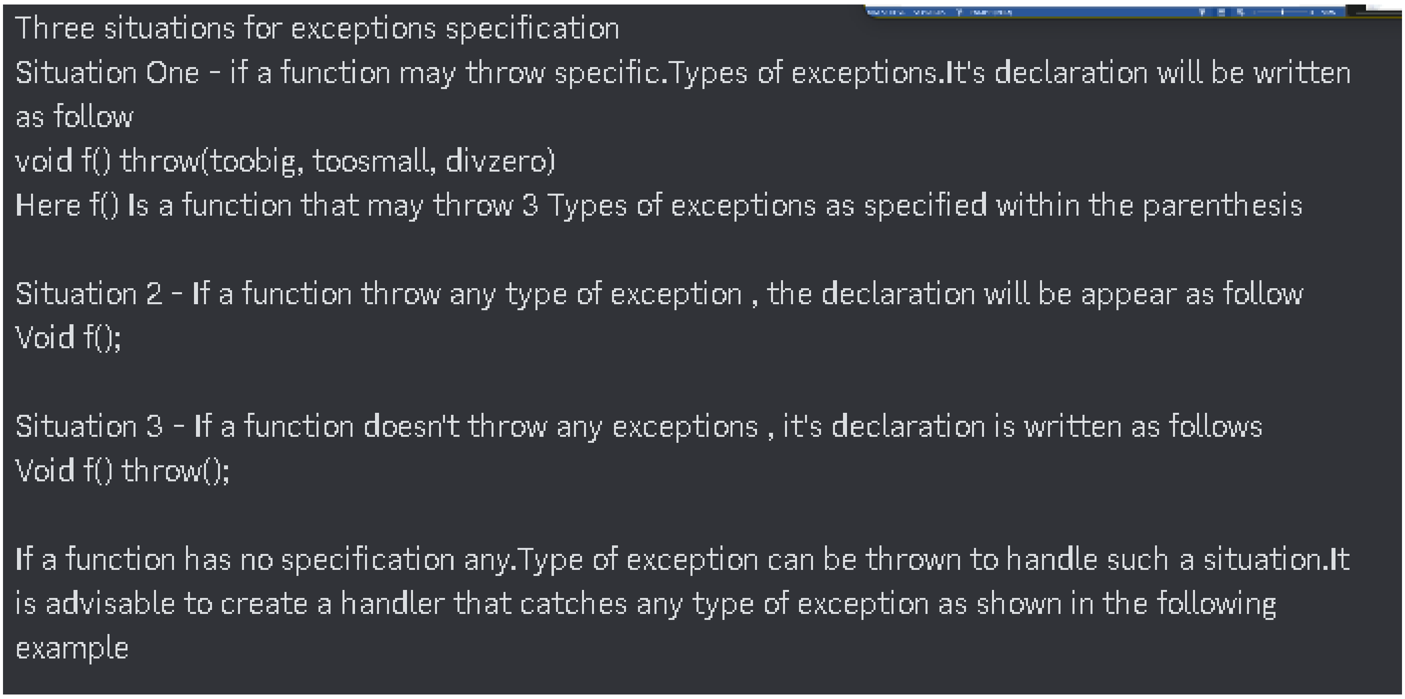


Sr. No.	Function overloading	Function template
3.	Function parameters are specified as C++ Data types.	Function parameters can be mixture of Template types and C++ types.
4.	Overloaded functions can have different number of parameters.	Since one function template is defined, number of parameters will remain same, while calling the function.
5.	Each overloaded function can have its own different logic and code.	Since there is one template function, same code works for various calls.
6.	Function definitions and calling has simple function syntax.	In function definition we need to mention 'template' data-type. Also, while calling the function we need mention actual data-type for which function template is called.
7.	Overloaded Functions can be called for specific data types, for which functions are defined.	Function template can be used for any compatible data types.
8.	Even if same algorithm is followed by all the overloaded functions, we need to define separate functions for different data types.	Only one function template is defined for working with different type of data.

2) What is an exception specification? Explain using suitable example.



Limits The exception that any function may throw during its run.
No exception specification means it can throw any exception.
An empty exception specification means it cannot throw an exception at all.



Three situations for exceptions specification
Situation One - if a function may throw specific.Types of exceptions.It's declaration will be written as follow

```
void f() throw(toobig, toosmall, divzero)
```

Here f() Is a function that may throw 3 Types of exceptions as specified within the parenthesis

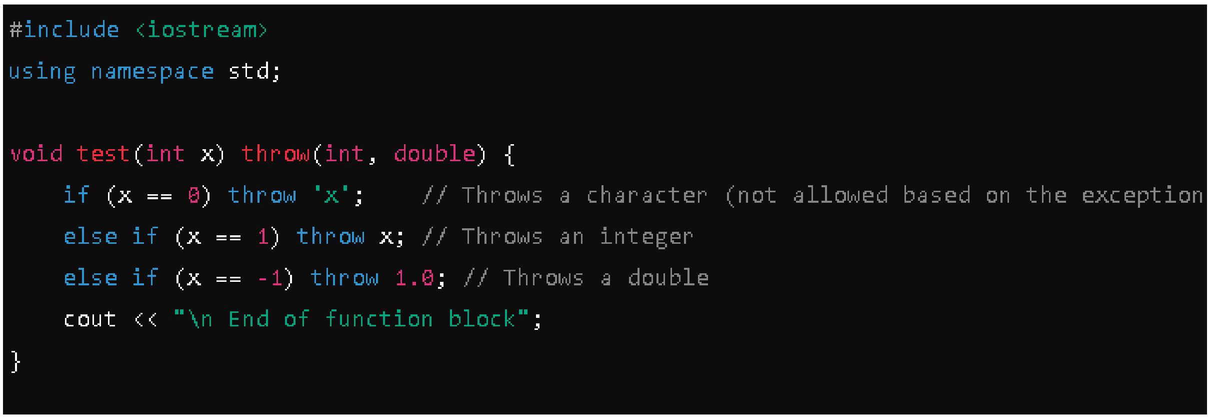
Situation 2 - If a function throw any type of exception , the declaration will be appear as follow

```
Void f();
```

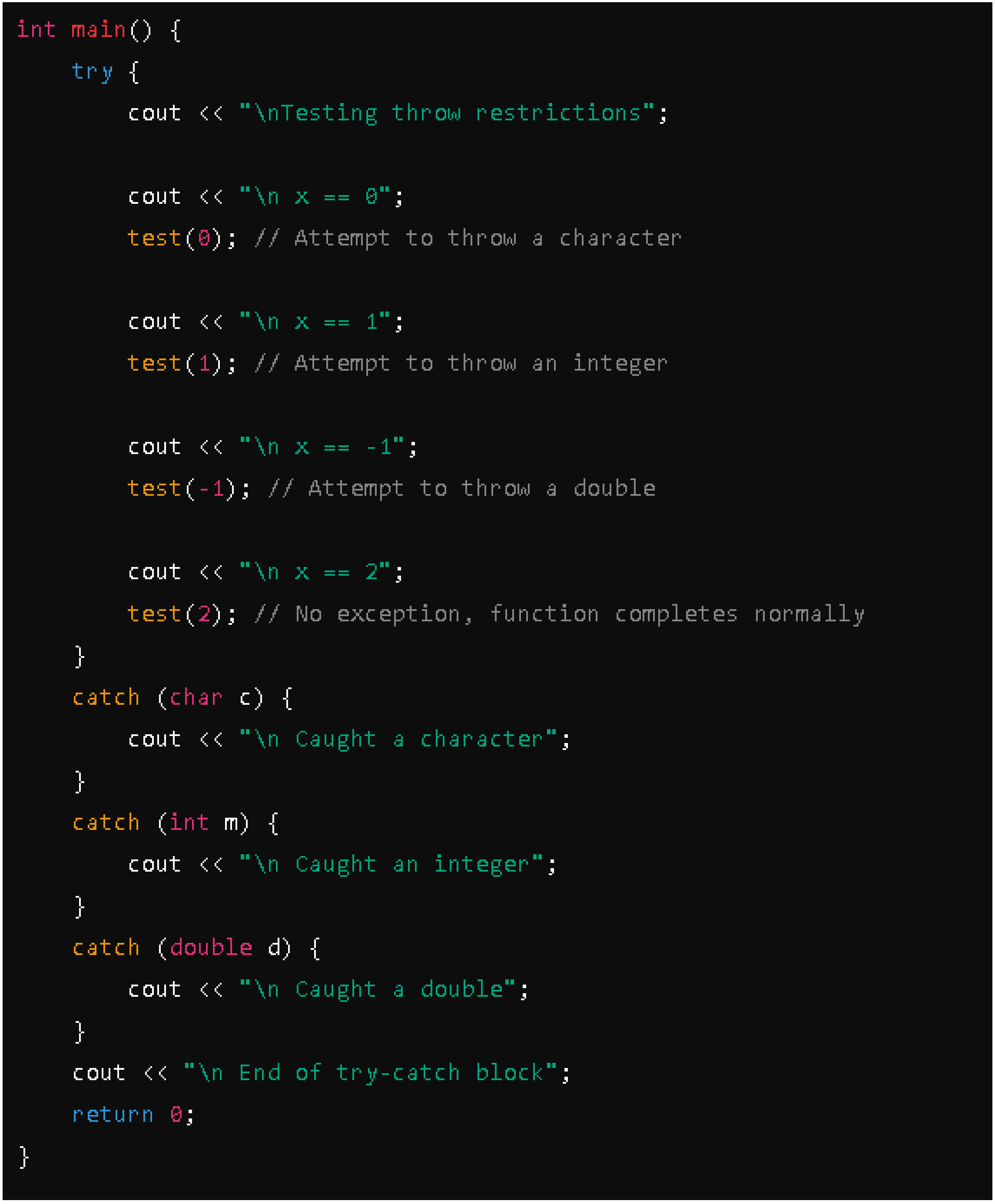
Situation 3 - If a function doesn't throw any exceptions , it's declaration is written as follows

```
Void f() throw();
```

If a function has no specification any.Type of exception can be thrown to handle such a situation.It is advisable to create a handler that catches any type of exception as shown in the following example

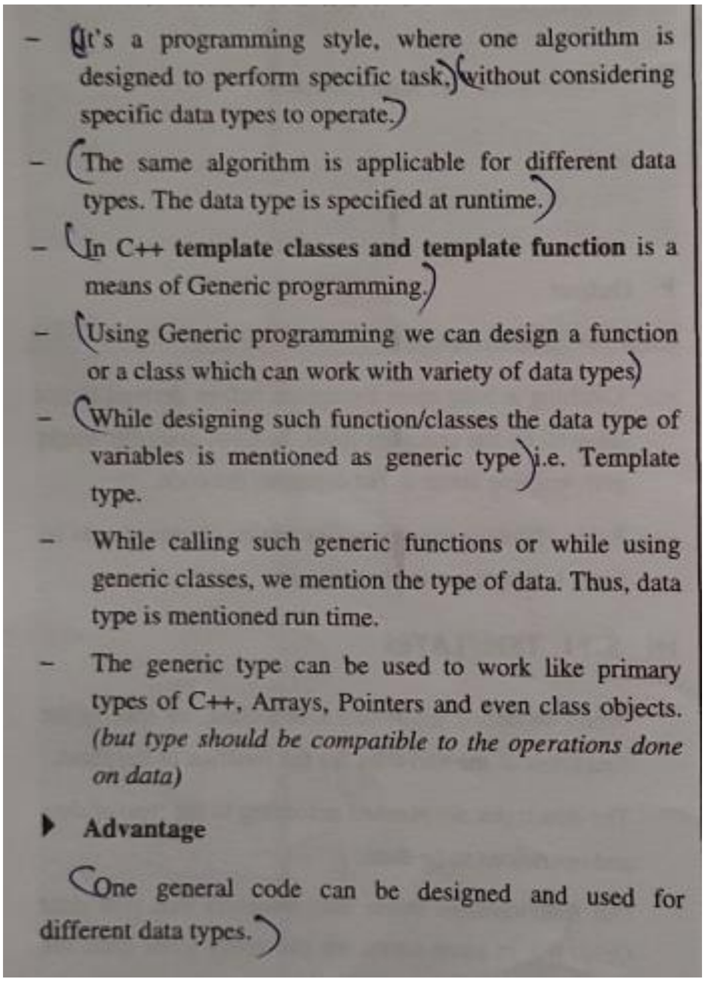


```cpp
#include <iostream>
using namespace std;

void test(int x) throw(int, double) {
    if (x == 0) throw 'x';    // Throws a character (not allowed based on the exception
    else if (x == 1) throw x; // Throws an integer
    else if (x == -1) throw 1.0; // Throws a double
    cout << "\n End of function block";
}
```



```cpp
int main() {
    try {
        cout << "\nTesting throw restrictions";

        cout << "\n x == 0";
        test(0); // Attempt to throw a character

        cout << "\n x == 1";
        test(1); // Attempt to throw an integer

        cout << "\n x == -1";
        test(-1); // Attempt to throw a double

        cout << "\n x == 2";
        test(2); // No exception, function completes normally
    }
    catch (char c) {
        cout << "\n Caught a character";
    }
    catch (int m) {
        cout << "\n Caught an integer";
    }
    catch (double d) {
        cout << "\n Caught a double";
    }
    cout << "\n End of try-catch block";
    return 0;
}
```

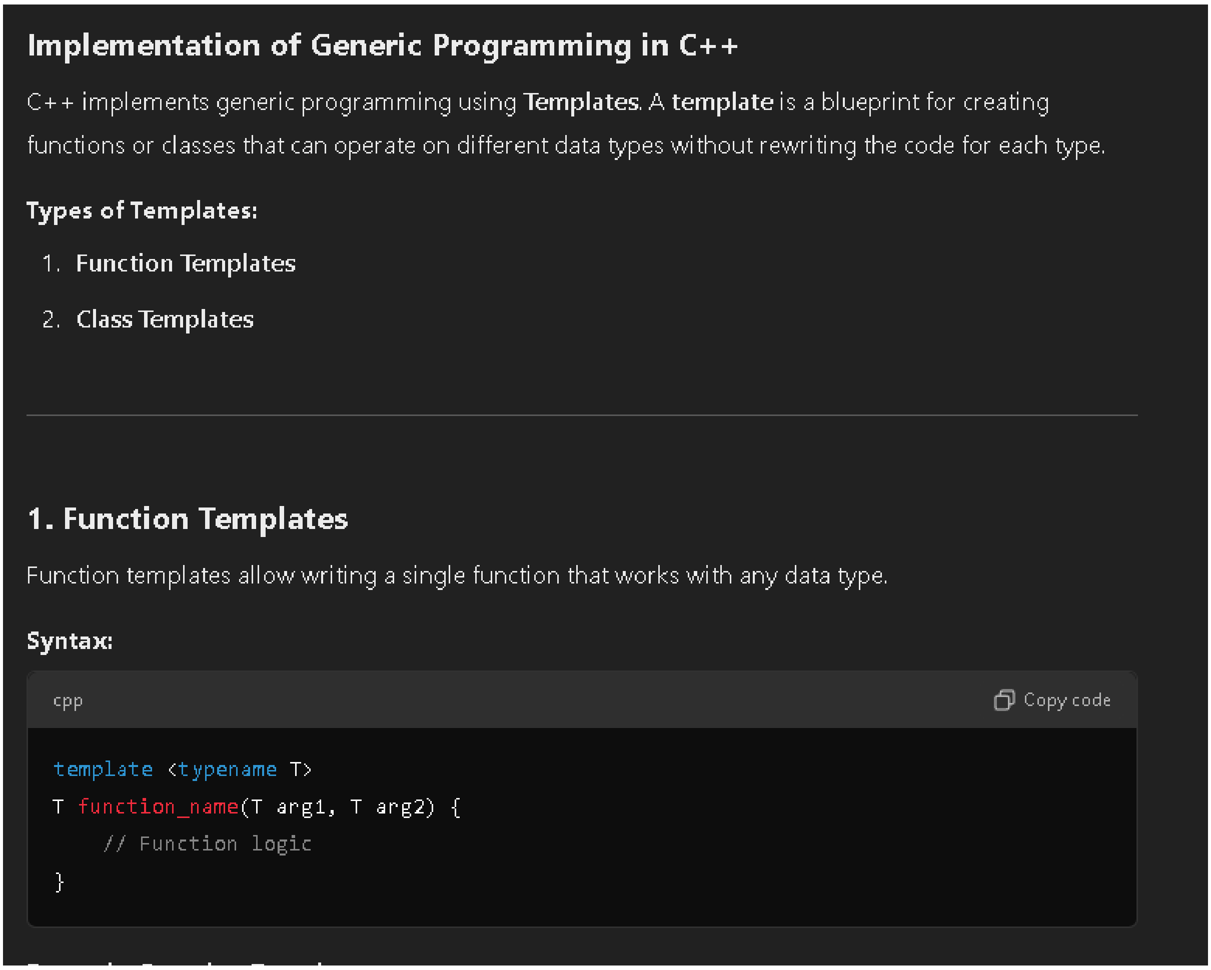
3) What is generic programming? How it is implemented in C++.



- It's a programming style, where one algorithm is designed to perform specific task, without considering specific data types to operate.
- The same algorithm is applicable for different data types. The data type is specified at runtime.
- In C++ **template classes and template function** is a means of Generic programming.
- Using Generic programming we can design a function or a class which can work with variety of data types.
- While designing such function/classes the data type of variables is mentioned as generic type i.e. Template type.
- While calling such generic functions or while using generic classes, we mention the type of data. Thus, data type is mentioned run time.
- The generic type can be used to work like primary types of C++, Arrays, Pointers and even class objects. *(but type should be compatible to the operations done on data)*

▶ **Advantage**

One general code can be designed and used for different data types.



Implementation of Generic Programming in C++

C++ implements generic programming using **Templates**. A **template** is a blueprint for creating functions or classes that can operate on different data types without rewriting the code for each type.

Types of Templates:

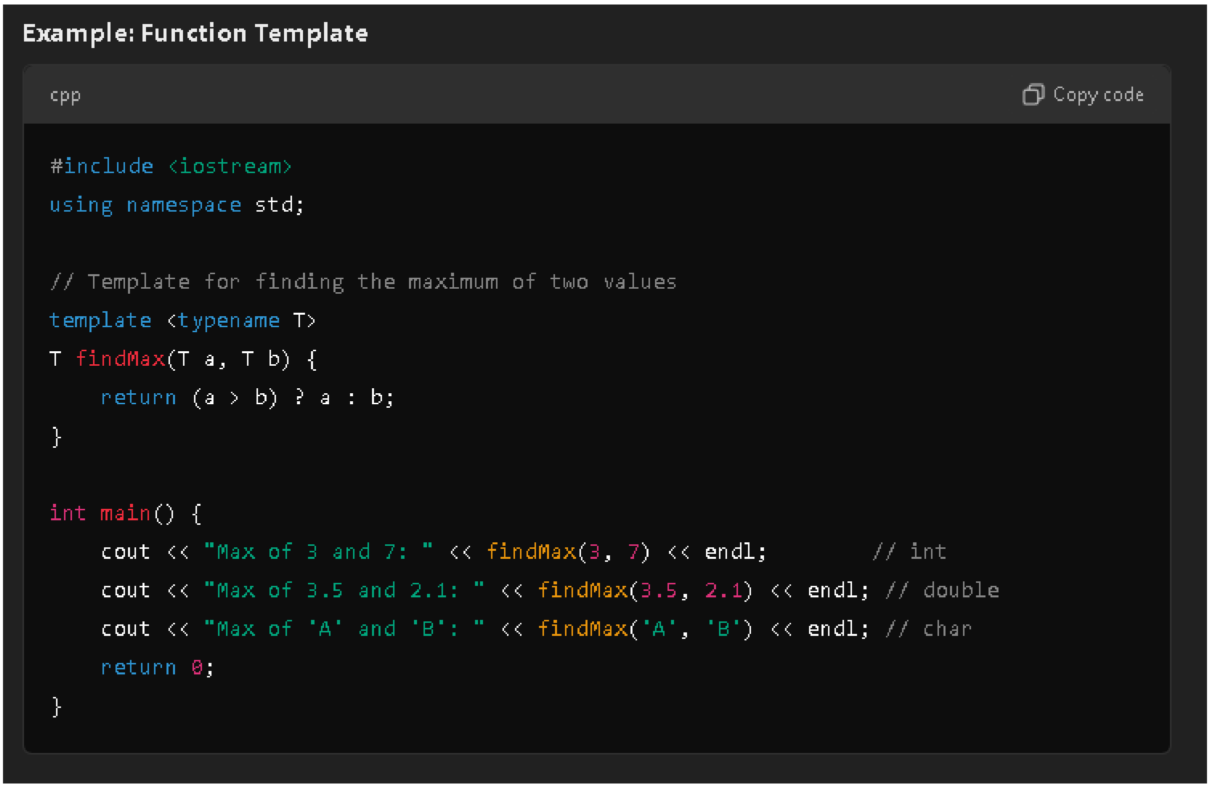
1. Function Templates
2. Class Templates

1. Function Templates

Function templates allow writing a single function that works with any data type.

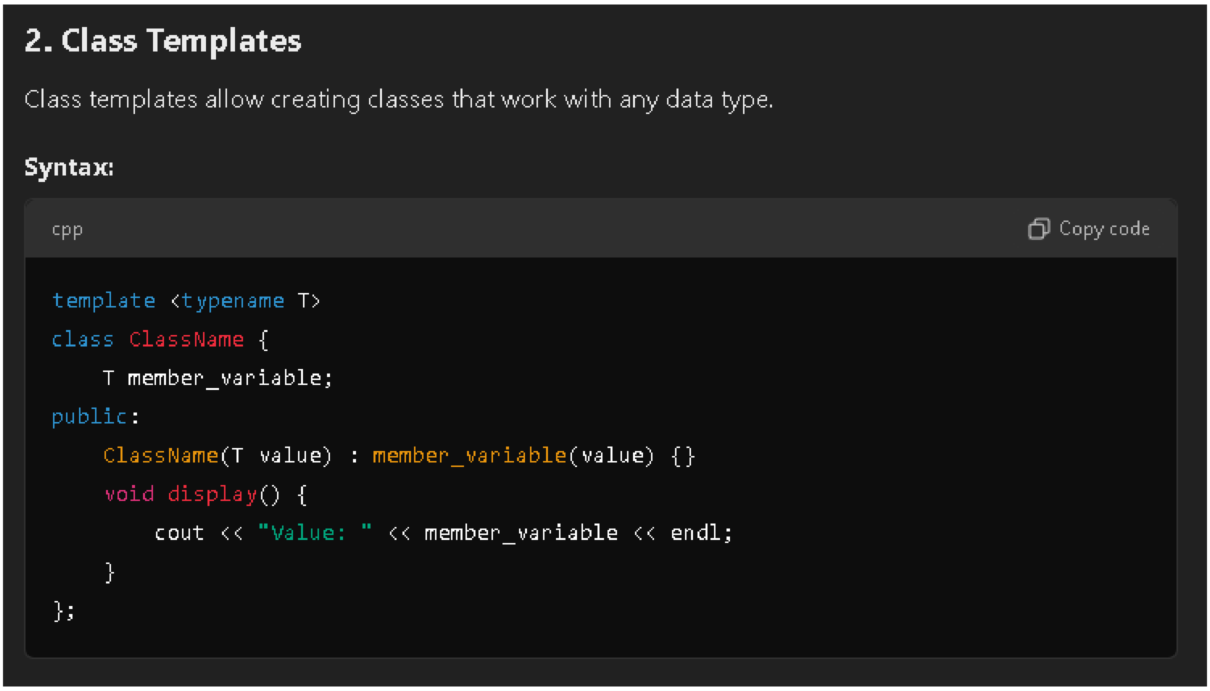
Syntax:

```cpp
template <typename T>
T function_name(T arg1, T arg2) {
    // Function logic
}
```



Example: Function Template

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

// Template for finding the maximum of two values
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max of 3 and 7: " << findMax(3, 7) << endl;        // int
    cout << "Max of 3.5 and 2.1: " << findMax(3.5, 2.1) << endl; // double
    cout << "Max of 'A' and 'B': " << findMax('A', 'B') << endl; // char
    return 0;
}
```

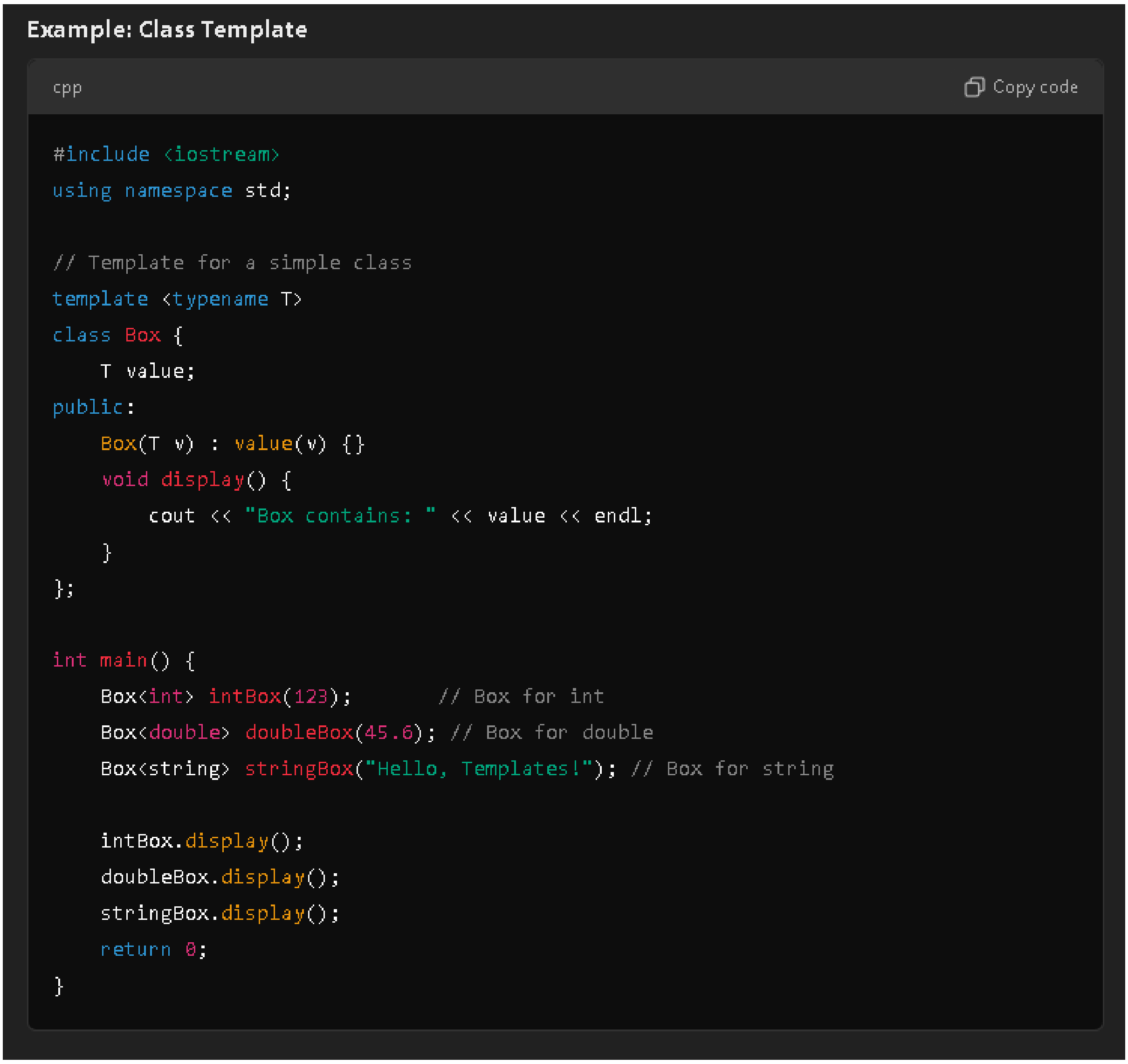


2. Class Templates

Class templates allow creating classes that work with any data type.

Syntax:

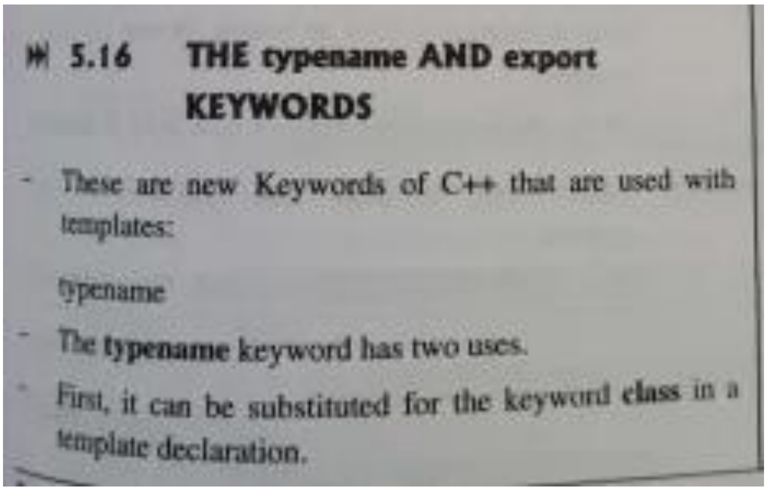
```cpp
template <typename T>
class ClassName {
    T member_variable;
public:
    ClassName(T value) : member_variable(value) {}
    void display() {
        cout << "Value: " << member_variable << endl;
    }
};
```



Example: Class Template

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

// Template for a simple class
template <typename T>
class Box {
    T value;
public:
    Box(T v) : value(v) {}
    void display() {
        cout << "Box contains: " << value << endl;
    }
};

int main() {
    Box<int> intBox(123);       // Box for int
    Box<double> doubleBox(45.6); // Box for double
    Box<string> stringBox("Hello, Templates!"); // Box for string

    intBox.display();
    doubleBox.display();
    stringBox.display();
    return 0;
}
```

4) Write short note on type name and export key-word

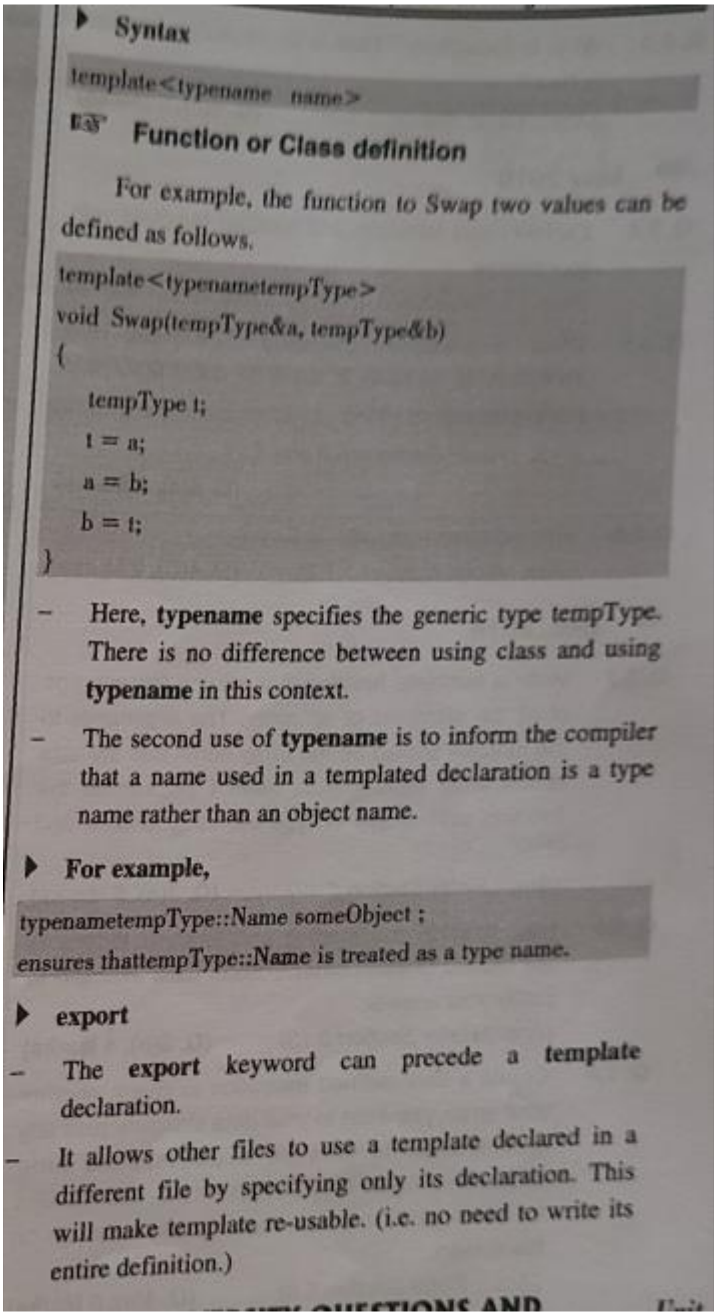


5.16 THE typename AND export KEYWORDS

- These are new Keywords of C++ that are used with templates:

typename

- The **typename** keyword has two uses.
- First, it can be substituted for the keyword **class** in a template declaration.



Syntax

```
template<typename name>
```

Function or Class definition

For example, the function to Swap two values can be defined as follows.

```
template<typenametempType>
void Swap(tempType&a, tempType&b)
{
   tempType t;
   t = a;
   a = b;
   b = t;
}
```

- Here, **typename** specifies the generic type tempType. There is no difference between using class and using **typename** in this context.
- The second use of **typename** is to inform the compiler that a name used in a templated declaration is a type name rather than an object name.

For example,

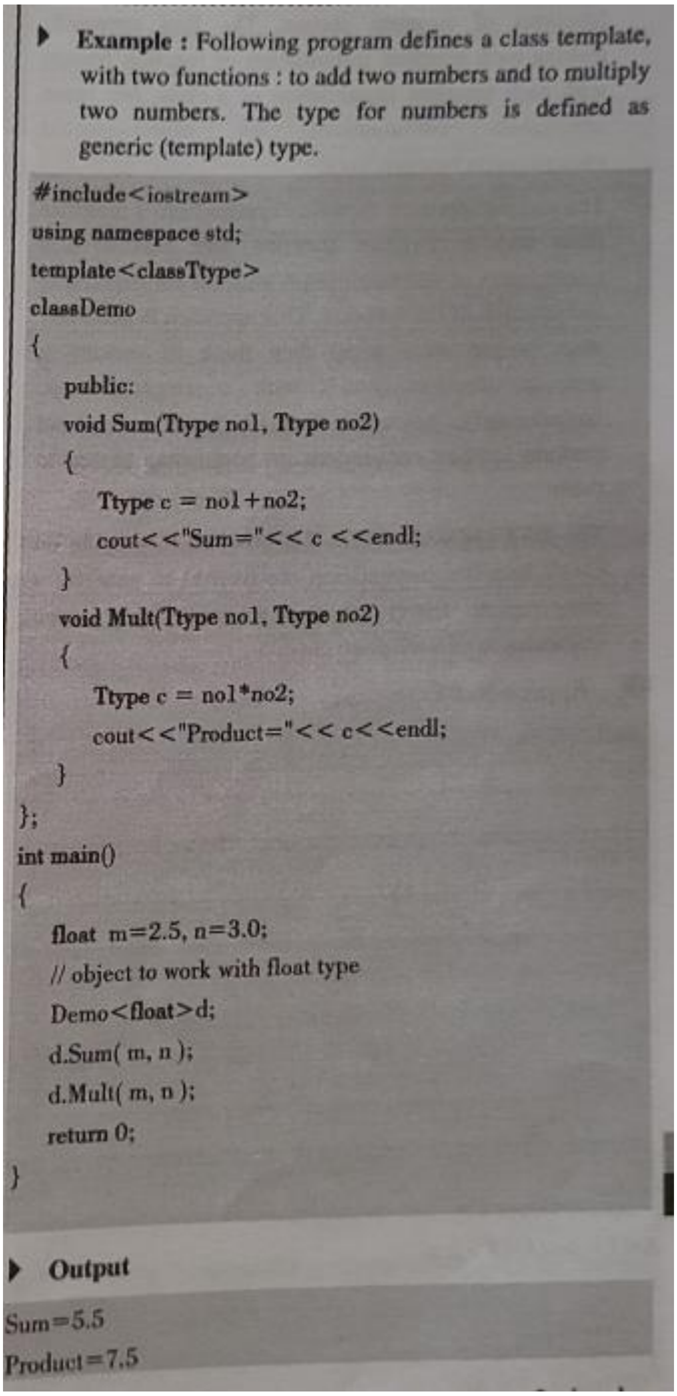
```
typenametempType::Name someObject ;
ensures thattempType::Name is treated as a type name.
```

export

- The **export** keyword can precede a **template** declaration.
- It allows other files to use a template declared in a different file by specifying only its declaration. This will make template re-usable. (i.e. no need to write its entire definition.)

QUESTIONS AND

5) Explain class template using multiple parameters with help of program

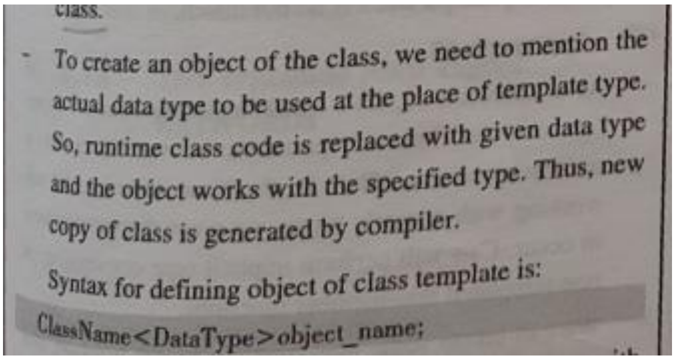


▶ **Example :** Following program defines a class template, with two functions : to add two numbers and to multiply two numbers. The type for numbers is defined as generic (template) type.

```
#include<iostream>
using namespace std;
template<classTtype>
classDemo
{
    public:
    void Sum(Ttype no1, Ttype no2)
    {
        Ttype c = no1+no2;
        cout<<"Sum="<< c <<endl;
    }
    void Mult(Ttype no1, Ttype no2)
    {
        Ttype c = no1*no2;
        cout<<"Product="<< c<<endl;
    }
};
int main()
{
    float m=2.5, n=3.0;
    // object to work with float type
    Demo<float>d;
    d.Sum( m, n );
    d.Mult( m, n );
    return 0;
}
```

▶ **Output**

```
Sum=5.5
Product=7.5
```

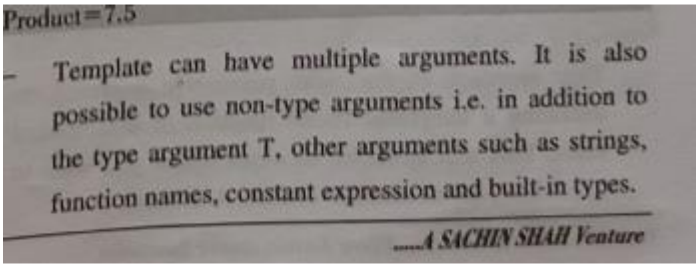


class.

- To create an object of the class, we need to mention the actual data type to be used at the place of template type. So, runtime class code is replaced with given data type and the object works with the specified type. Thus, new copy of class is generated by compiler.

Syntax for defining object of class template is:

```
ClassName<DataType>object_name;
```

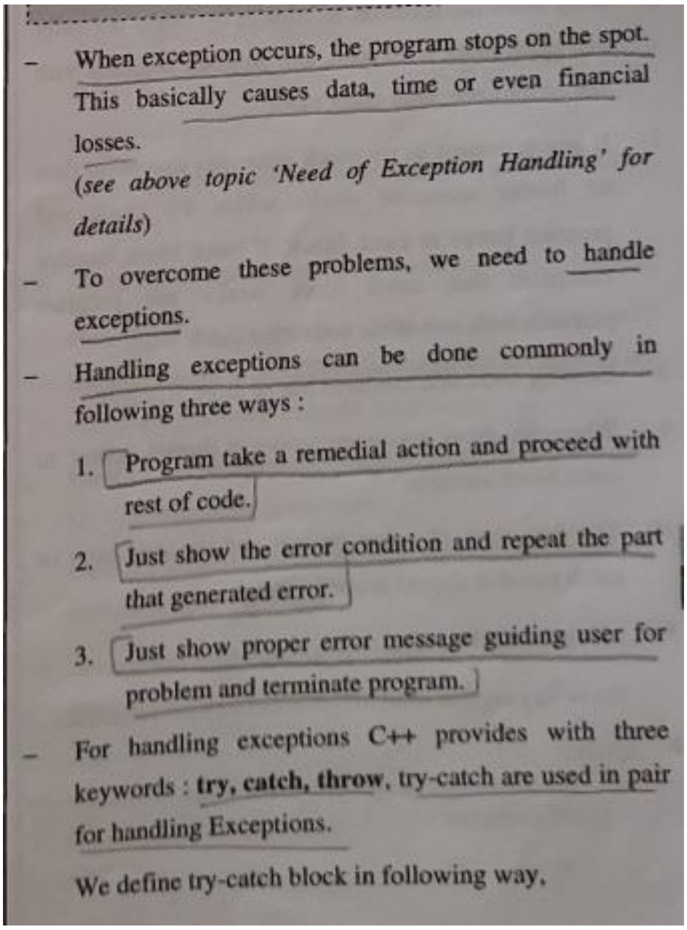


```
Product=7.5
```

- Template can have multiple arguments. It is also possible to use non-type arguments i.e. in addition to the type argument T, other arguments such as strings, function names, constant expression and built-in types.

.....A SACHIN SHAH Venture

6) Explain exception handling mechanism in C++? Explain by program to handle "divide by zero"



- When exception occurs, the program stops on the spot. This basically causes data, time or even financial losses.

  (see above topic 'Need of Exception Handling' for details)

- To overcome these problems, we need to handle exceptions.
- Handling exceptions can be done commonly in following three ways :

  1. Program take a remedial action and proceed with rest of code.
  2. Just show the error condition and repeat the part that generated error.
  3. Just show proper error message guiding user for problem and terminate program.

- For handling exceptions C++ provides with three keywords : **try, catch, throw**. try-catch are used in pair for handling Exceptions.

  We define try-catch block in following way,

```
try
{
    // code that may generate exception
}
catch( Type variable )
{
// Code to handle exception
}
```

1. The 'try' block consists of code that may generate exception.
2. The catch block traps any error thrown (generated) by code in try block.
3. The 'Type' is any primary or user defined type.
4. We can write multiple catch blocks with one try.
5. We can nest one try-catch block in another try block.
6. If try block has no error, all code in try works normally and catch block is skipped, and program proceeds with rest of the code, after catch block.
7. If any statement in try block generates exception then no further statement works within try block and program jumps to catch block. If catch block handles exception then catch block works and program proceeds with rest of the code after catch block.
8. Each try and catch can have one or more statements.
9. When the above statement works, it throws 'value' to catch block variable.
10. The data type of value thrown and the data type of catch variable should properly match.

☞ Keyword throw

throw keyword is used to throw error in the program.

▶ Syntax

```
    throw exception ;
```

Zero

> **GQ. 5.1.6** *Write a program to demonstrate how to handle 'divide by zero' exception.*
>
> **UQ. 5.1.7** *Write a program in C++ to handle "divide by zero" exception.*
>
> **SPPU - Q. 4(b), Dec. 16, 3 Marks**

- Following program input two numbers.
- If second number is zero then program throws (i.e. generates) exception otherwise it prints answer of division.

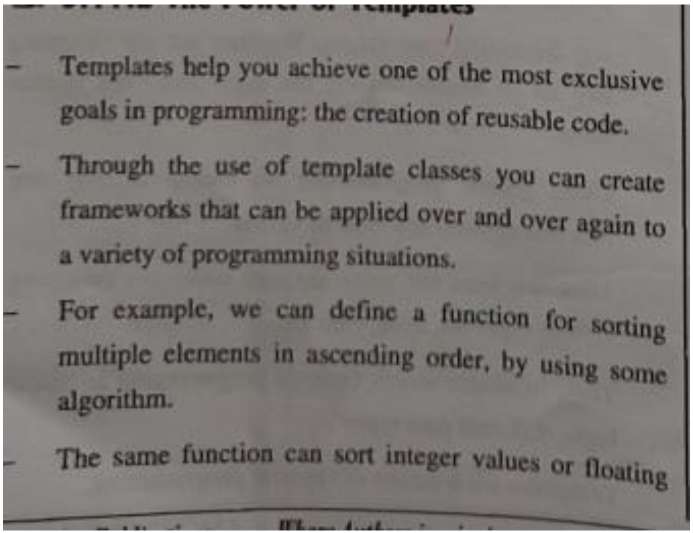
☞ Program Code

```
#include<iostream>
using namespace std;
int main()
{
   float a, b, c;
   try
   {
      cout<<"Enter two numbers:";
      cin>> a >> b;
      if( b == 0)
         throw b;
      else
         c = a/b;
      cout<<"Ans = "<< c <<endl;
   }
   catch( float ex )
   {
      cout<<"Div. by Error"<<endl;
   }
   cout<<"Program ends\n";
   return 0;
}
```

▶ Output

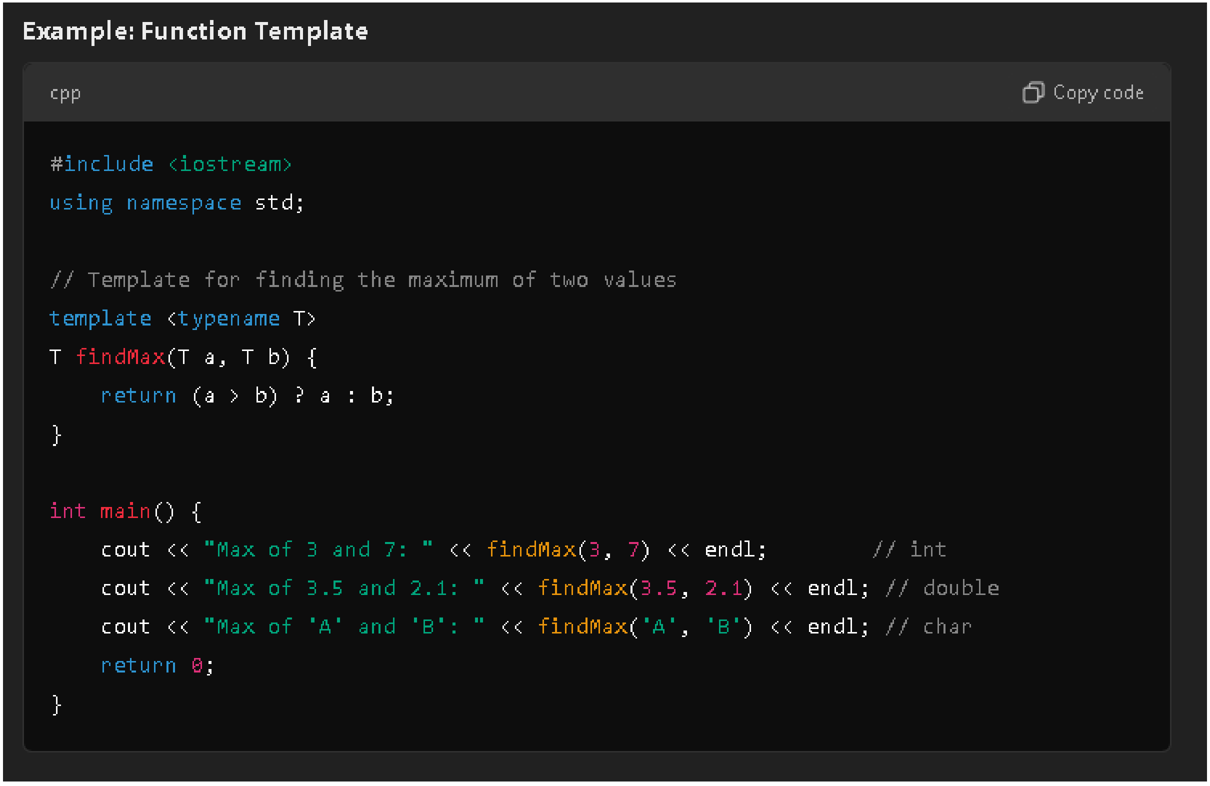
```
Enter two numbers:5
0
Div. by Error
Program ends
```

7) What is the power of templates in C++? Explain along with one example



The Power of Templates

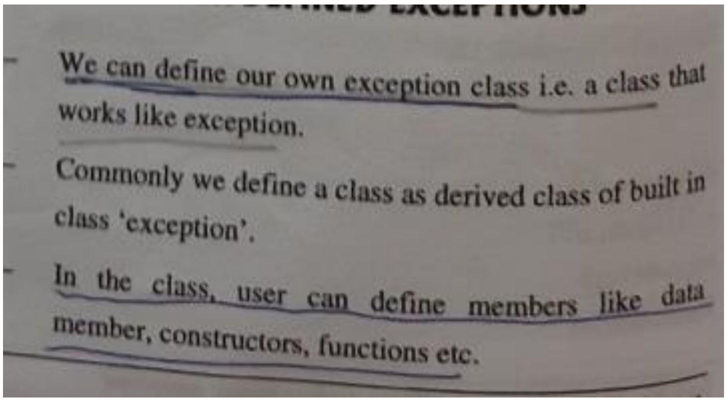
- Templates help you achieve one of the most exclusive goals in programming: the creation of reusable code.
- Through the use of template classes you can create frameworks that can be applied over and over again to a variety of programming situations.
- For example, we can define a function for sorting multiple elements in ascending order, by using some algorithm.
- The same function can sort integer values or floating



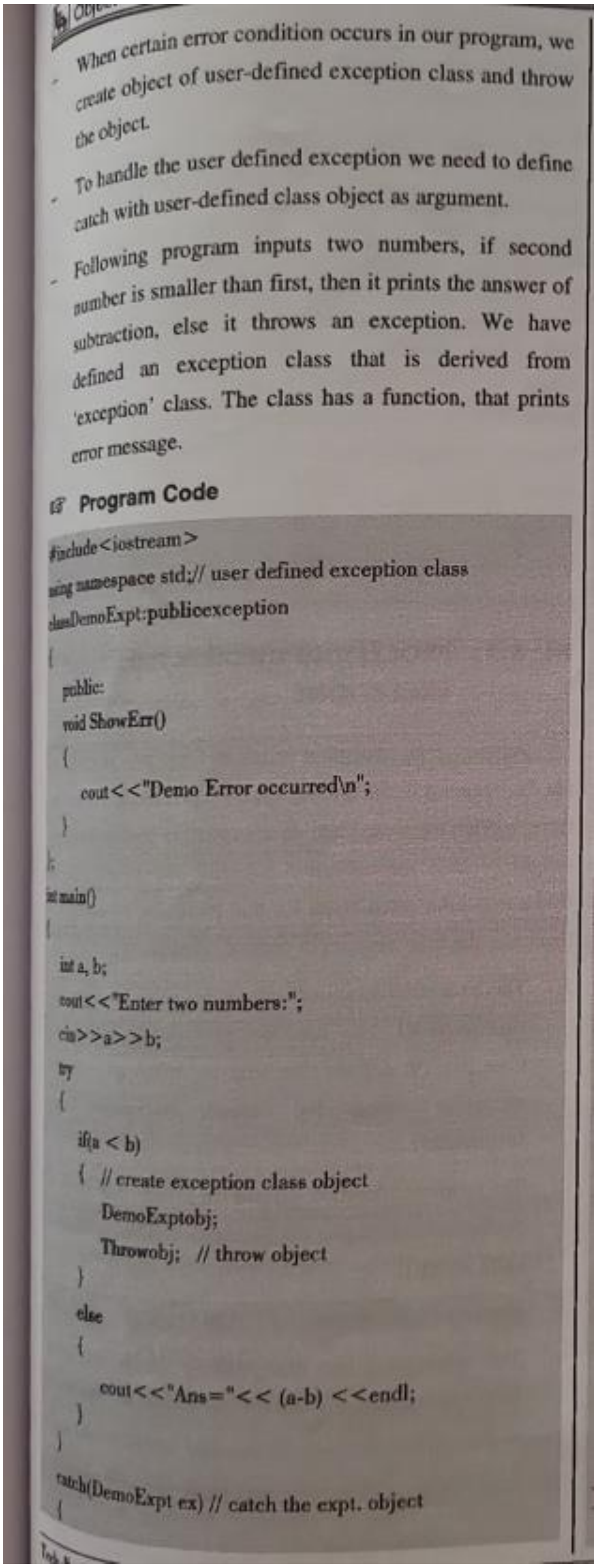
Example: Function Template

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

// Template for finding the maximum of two values
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max of 3 and 7: " << findMax(3, 7) << endl;        // int
    cout << "Max of 3.5 and 2.1: " << findMax(3.5, 2.1) << endl; // double
    cout << "Max of 'A' and 'B': " << findMax('A', 'B') << endl; // char
    return 0;
}
```

8) What is mean by user defined exception? Give one example.



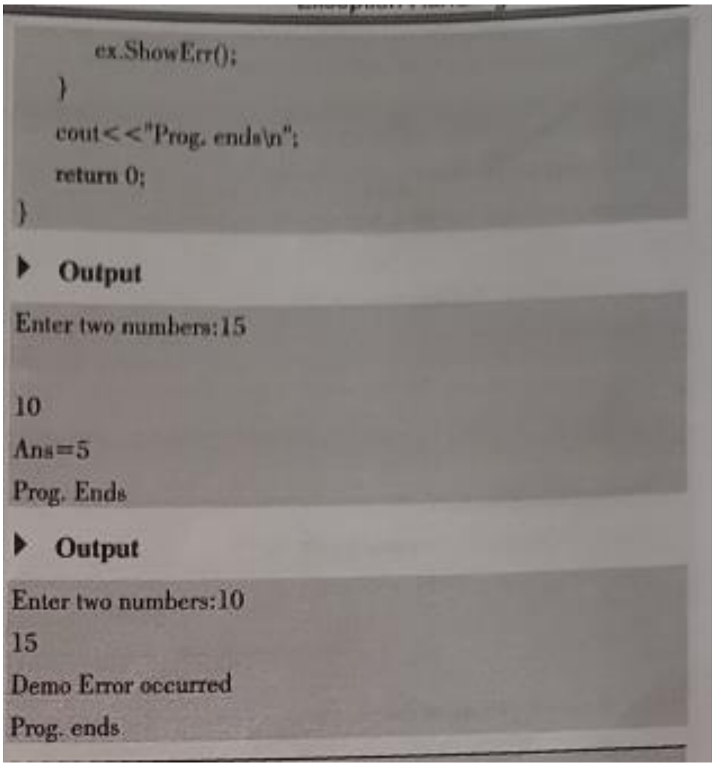
- We can define our own exception class i.e. a class that works like exception.
- Commonly we define a class as derived class of built in class 'exception'.
- In the class, user can define members like data member, constructors, functions etc.



- When certain error condition occurs in our program, we create object of user-defined exception class and throw the object.
- To handle the user defined exception we need to define catch with user-defined class object as argument.
- Following program inputs two numbers, if second number is smaller than first, then it prints the answer of subtraction, else it throws an exception. We have defined an exception class that is derived from 'exception' class. The class has a function, that prints error message.

Program Code

```
#include<iostream>
using namespace std;// user defined exception class
classDemoExpt:publicexception
{
 public:
 void ShowErr()
 {
   cout<<"Demo Error occurred\n";
 }
};
int main()
{
 int a, b;
 cout<<"Enter two numbers:";
 cin>>a>>b;
 try
 {
   if(a < b)
   { // create exception class object
     DemoExptobj;
     Throwobj;  // throw object
   }
   else
   {
     cout<<"Ans="<< (a-b) <<endl;
   }
 }
 catch(DemoExpt ex) // catch the expt. object
 {
```



```
     ex.ShowErr();
 }
 cout<<"Prog. ends\n";
 return 0;
}
```

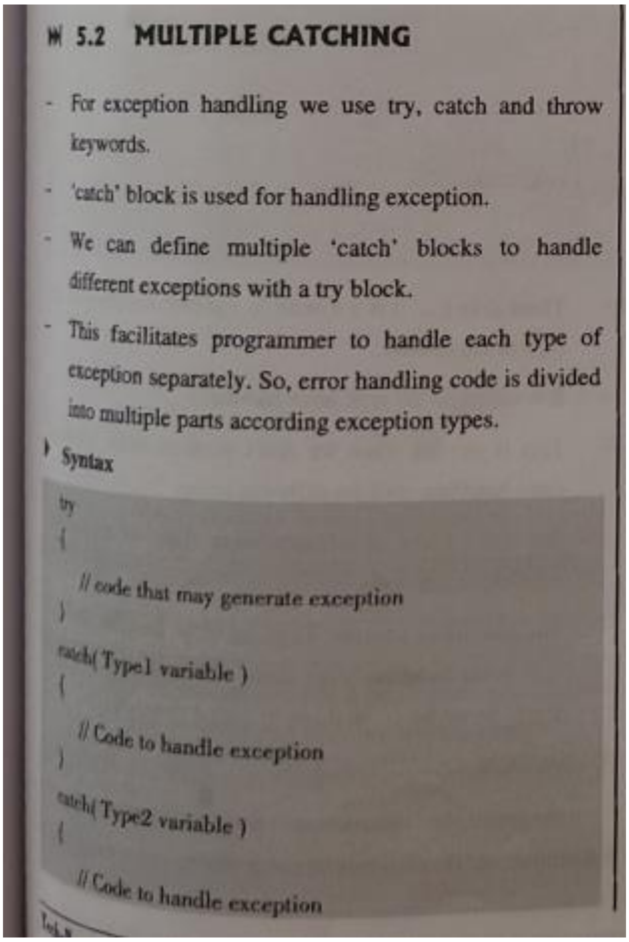
Output

```
Enter two numbers:15

10
Ans=5
Prog. Ends
```

Output

```
Enter two numbers:10
15
Demo Error occurred
Prog. ends
```

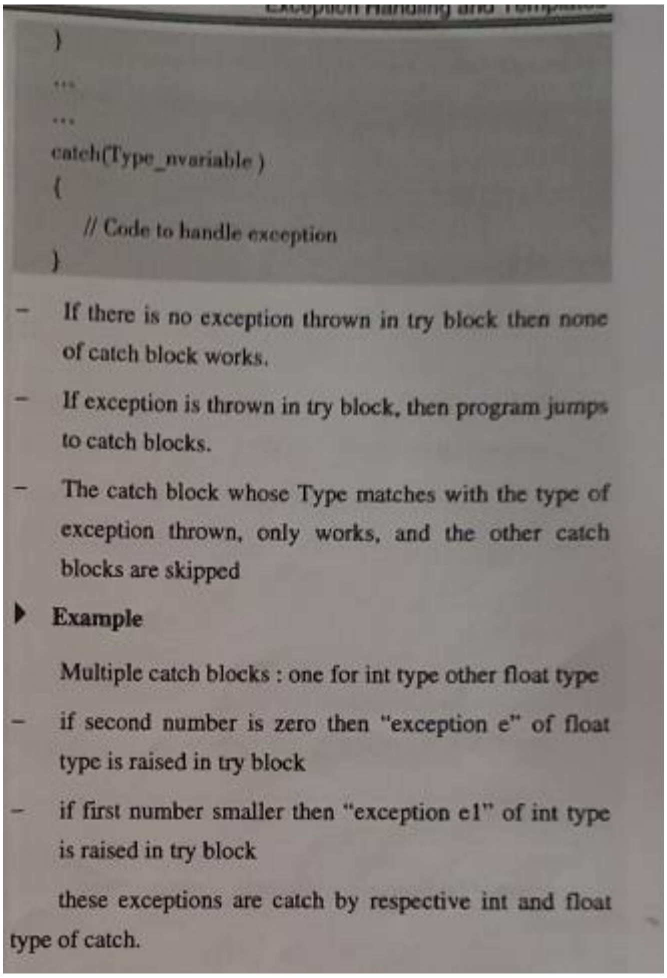
9) How multiple catching is implemented in exception handling?



5.2 MULTIPLE CATCHING

- For exception handling we use try, catch and throw keywords.
- 'catch' block is used for handling exception.
- We can define multiple 'catch' blocks to handle different exceptions with a try block.
- This facilitates programmer to handle each type of exception separately. So, error handling code is divided into multiple parts according exception types.

Syntax



```
try
{
    // code that may generate exception
}
catch( Type1 variable )
{
    // Code to handle exception
}
catch( Type2 variable )
{
    // Code to handle exception
}
...
...
catch(Type_nvariable )
{
    // Code to handle exception
}
```

- If there is no exception thrown in try block then none of catch block works.
- If exception is thrown in try block, then program jumps to catch blocks.
- The catch block whose Type matches with the type of exception thrown, only works, and the other catch blocks are skipped

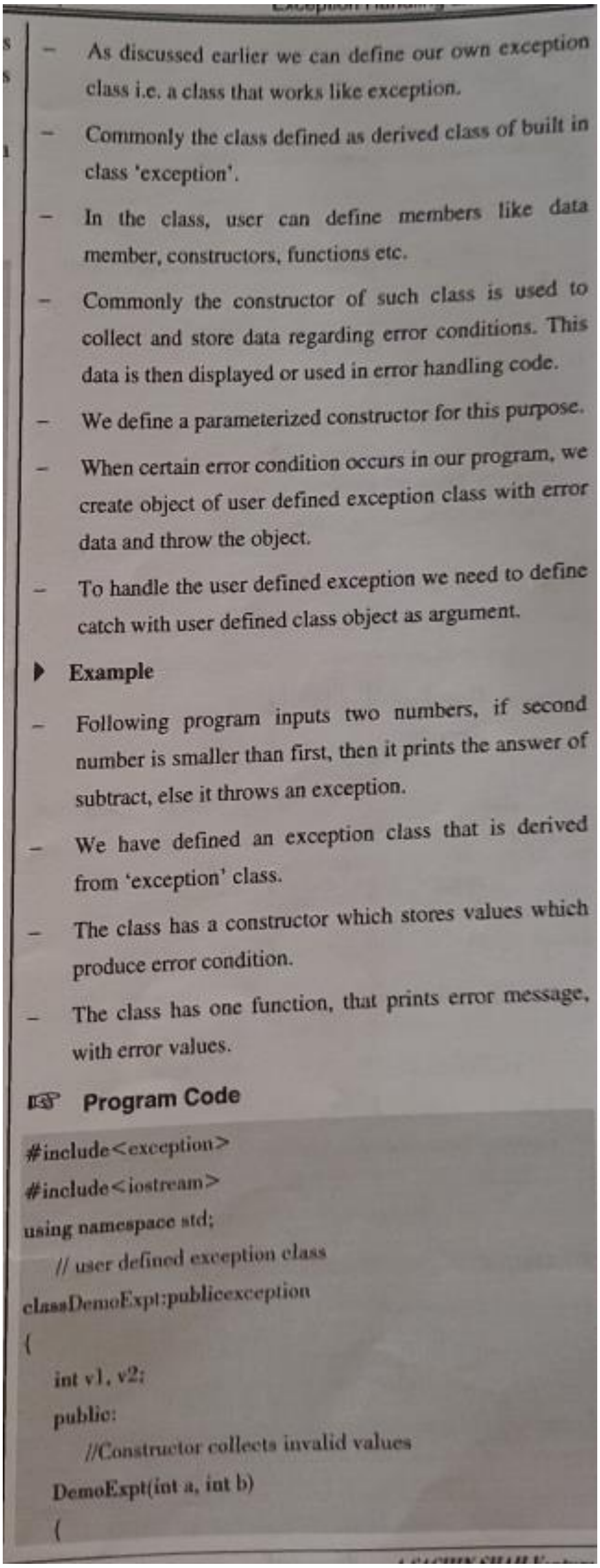
Example

Multiple catch blocks : one for int type other float type

- if second number is zero then "exception e" of float type is raised in try block
- if first number smaller then "exception e1" of int type is raised in try block

these exceptions are catch by respective int and float type of catch.

10) Explain exception handling in constructor, destructor.



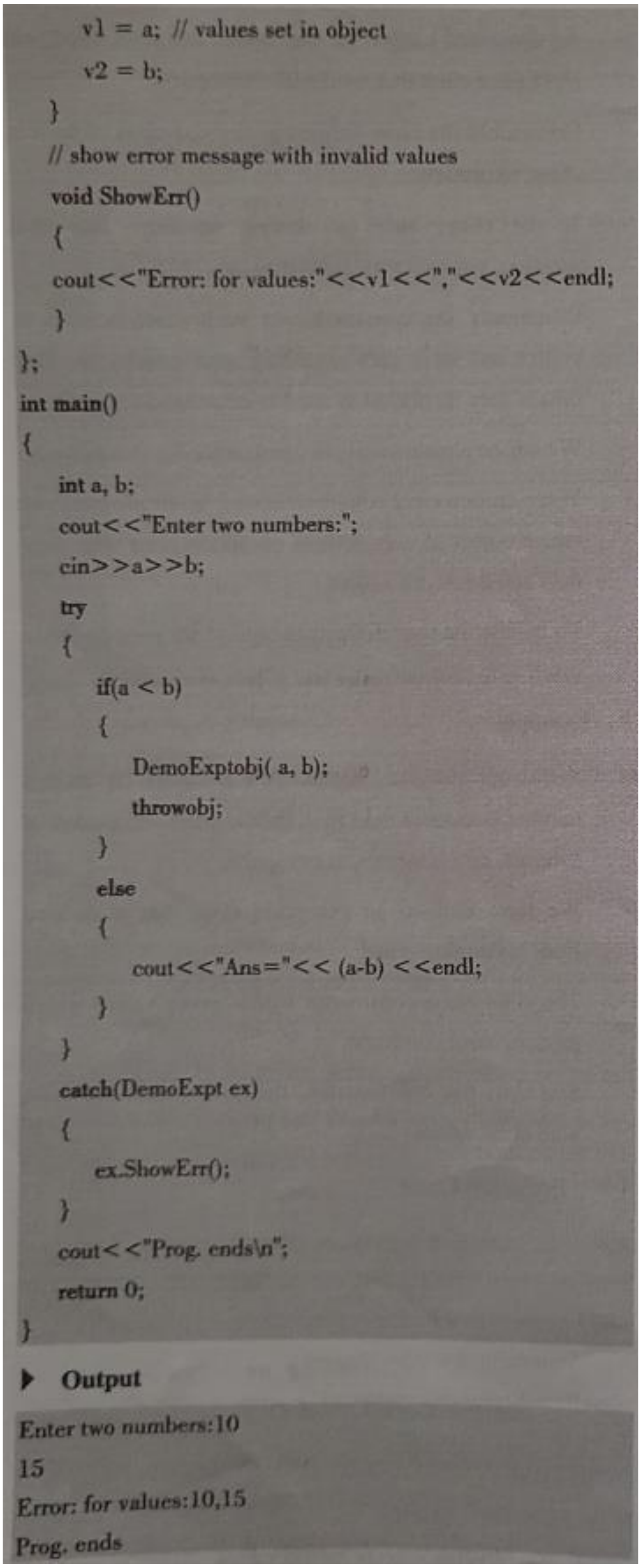
- As discussed earlier we can define our own exception class i.e. a class that works like exception.
- Commonly the class defined as derived class of built in class 'exception'.
- In the class, user can define members like data member, constructors, functions etc.
- Commonly the constructor of such class is used to collect and store data regarding error conditions. This data is then displayed or used in error handling code.
- We define a parameterized constructor for this purpose.
- When certain error condition occurs in our program, we create object of user defined exception class with error data and throw the object.
- To handle the user defined exception we need to define catch with user defined class object as argument.

▶ **Example**

- Following program inputs two numbers, if second number is smaller than first, then it prints the answer of subtract, else it throws an exception.
- We have defined an exception class that is derived from 'exception' class.
- The class has a constructor which stores values which produce error condition.
- The class has one function, that prints error message, with error values.

☞ **Program Code**

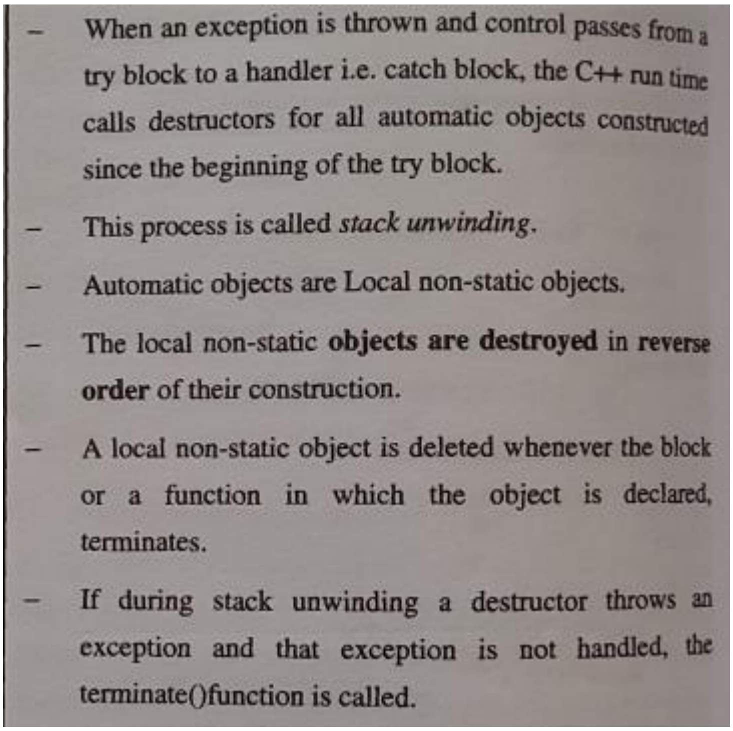
```
#include<exception>
#include<iostream>
using namespace std;
   // user defined exception class
classDemoExpt:publicexception
{
   int v1, v2;
   public:
      //Constructor collects invalid values
   DemoExpt(int a, int b)
   {
```



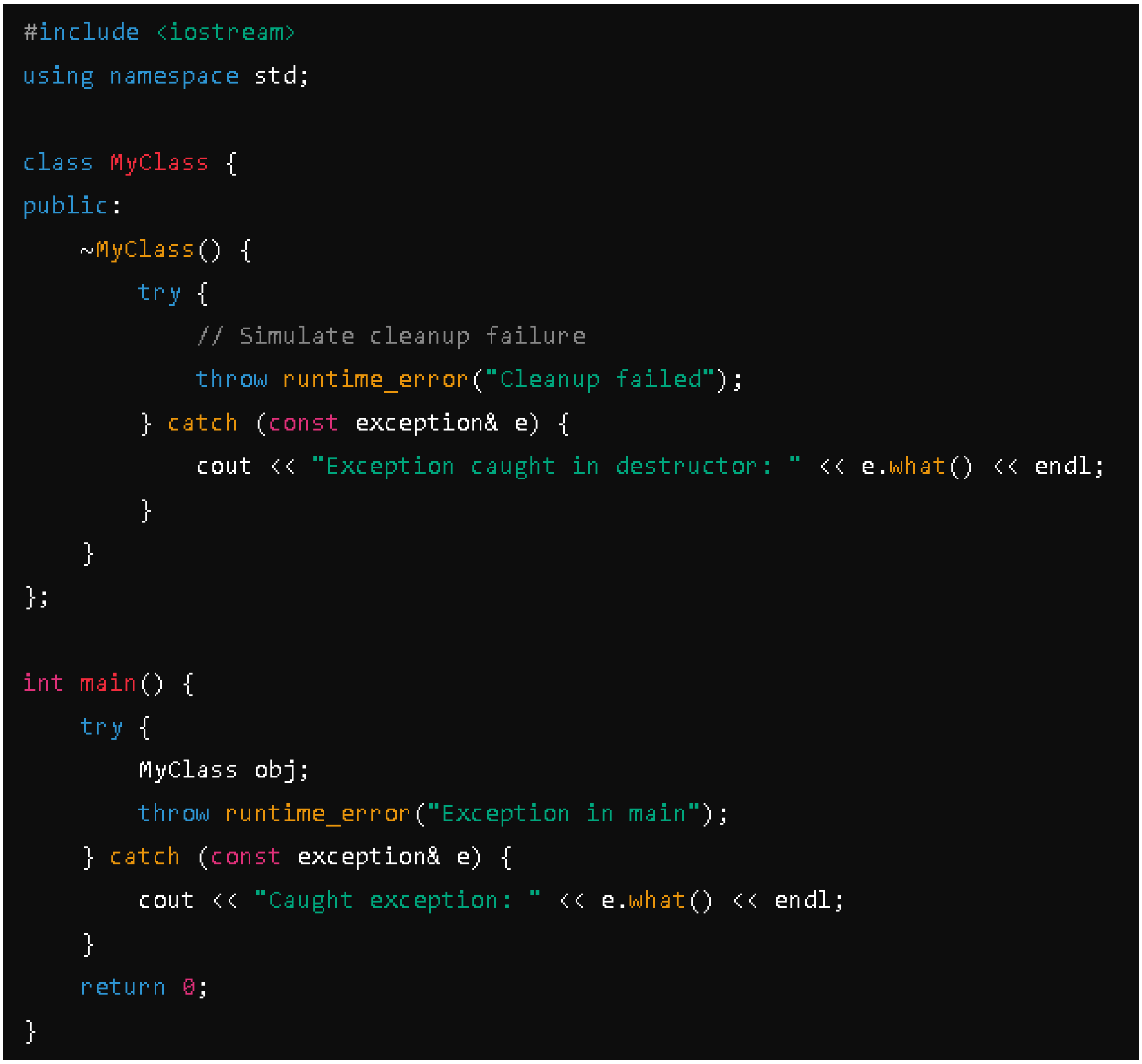
```
      v1 = a; // values set in object
      v2 = b;
   }
   // show error message with invalid values
   void ShowErr()
   {
   cout<<"Error: for values:"<<v1<<","<<v2<<endl;
   }
};
int main()
{
   int a, b;
   cout<<"Enter two numbers:";
   cin>>a>>b;
   try
   {
      if(a < b)
      {
         DemoExptobj( a, b);
         throwobj;
      }
      else
      {
         cout<<"Ans="<< (a-b) <<endl;
      }
   }
   catch(DemoExpt ex)
   {
      ex.ShowErr();
   }
   cout<<"Prog. ends\n";
   return 0;
}
```

▶ **Output**

```
Enter two numbers:10
15
Error: for values:10,15
Prog. ends
```

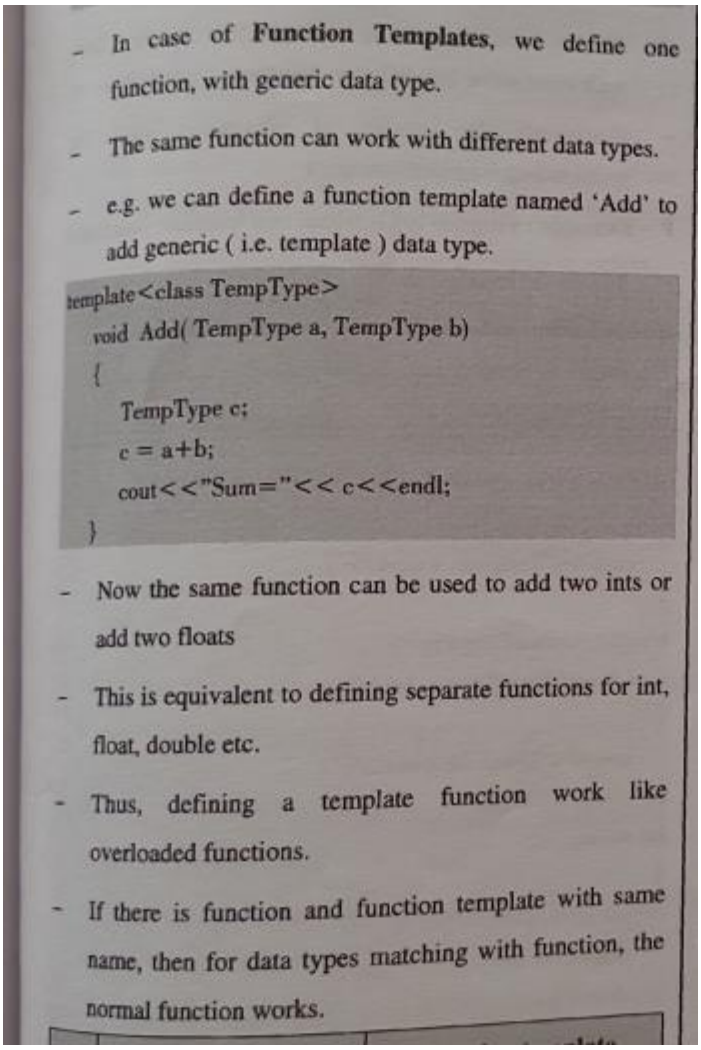


- When an exception is thrown and control passes from a try block to a handler i.e. catch block, the C++ run time calls destructors for all automatic objects constructed since the beginning of the try block.
- This process is called *stack unwinding*.
- Automatic objects are Local non-static objects.
- The local non-static **objects are destroyed** in **reverse order** of their construction.
- A local non-static object is deleted whenever the block or a function in which the object is declared, terminates.
- If during stack unwinding a destructor throws an exception and that exception is not handled, the terminate()function is called.



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

class MyClass {
public:
    ~MyClass() {
        try {
            // Simulate cleanup failure
            throw runtime_error("Cleanup failed");
        } catch (const exception& e) {
            cout << "Exception caught in destructor: " << e.what() << endl;
        }
    }
};

int main() {
    try {
        MyClass obj;
        throw runtime_error("Exception in main");
    } catch (const exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
```

11) Demonstrate overloading function template with suitable code in C++

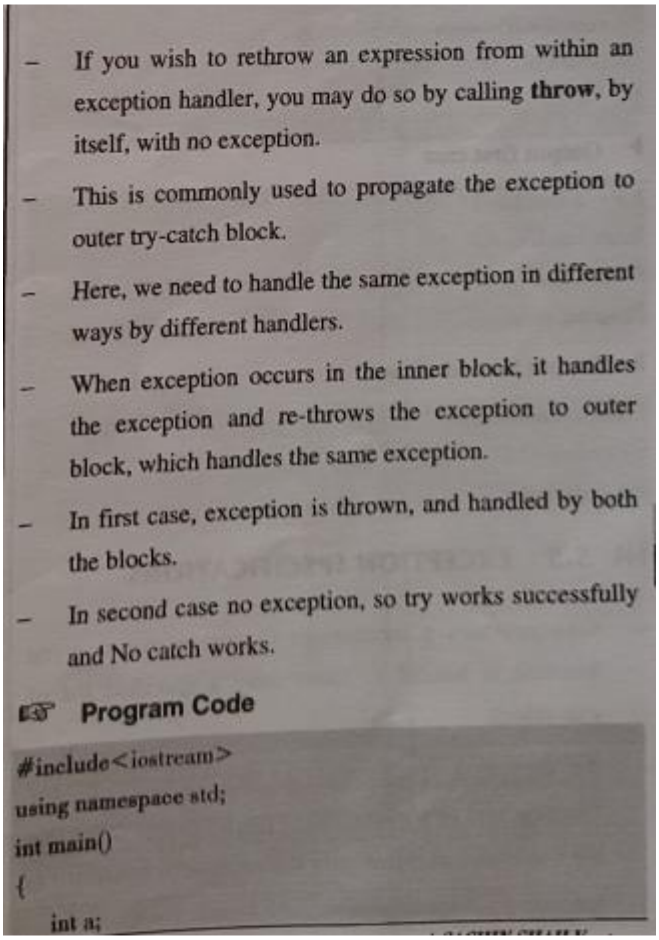


- In case of **Function Templates**, we define one function, with generic data type.
- The same function can work with different data types.
- e.g. we can define a function template named 'Add' to add generic (i.e. template) data type.

```
template<class TempType>
  void Add( TempType a, TempType b)
  {
     TempType c;
     c = a+b;
     cout<<"Sum="<< c<<endl;
  }
```

- Now the same function can be used to add two ints or add two floats
- This is equivalent to defining separate functions for int, float, double etc.
- Thus, defining a template function work like overloaded functions.
- If there is function and function template with same name, then for data types matching with function, the normal function works.

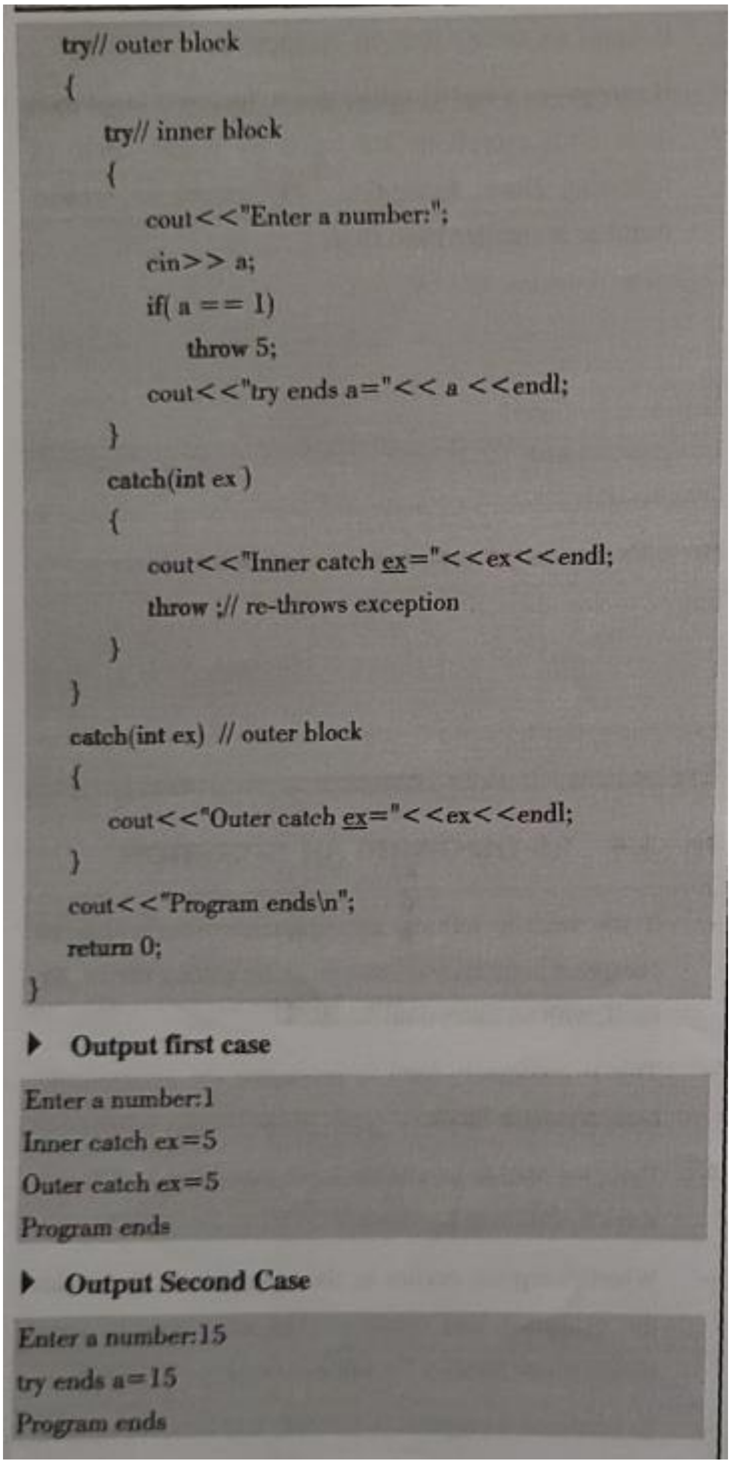
12) What do you mean by rethrowing exceptions. Write a program for the same



- If you wish to rethrow an expression from within an exception handler, you may do so by calling **throw**, by itself, with no exception.
- This is commonly used to propagate the exception to outer try-catch block.
- Here, we need to handle the same exception in different ways by different handlers.
- When exception occurs in the inner block, it handles the exception and re-throws the exception to outer block, which handles the same exception.
- In first case, exception is thrown, and handled by both the blocks.
- In second case no exception, so try works successfully and No catch works.

Program Code

```
#include<iostream>
using namespace std;
int main()
{
    int a;
```



```
    try// outer block
    {
        try// inner block
        {
            cout<<"Enter a number:";
            cin>> a;
            if( a == 1)
                throw 5;
            cout<<"try ends a="<< a <<endl;
        }
        catch(int ex )
        {
            cout<<"Inner catch ex="<<ex<<endl;
            throw ;// re-throws exception
        }
    }
    catch(int ex)  // outer block
    {
        cout<<"Outer catch ex="<<ex<<endl;
    }
    cout<<"Program ends\n";
    return 0;
}
```

Output first case

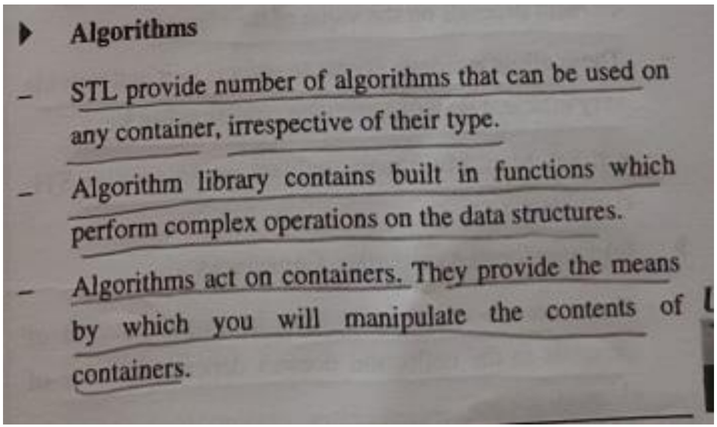
```
Enter a number:1
Inner catch ex=5
Outer catch ex=5
Program ends
```

Output Second Case

```
Enter a number:15
try ends a=15
Program ends
```

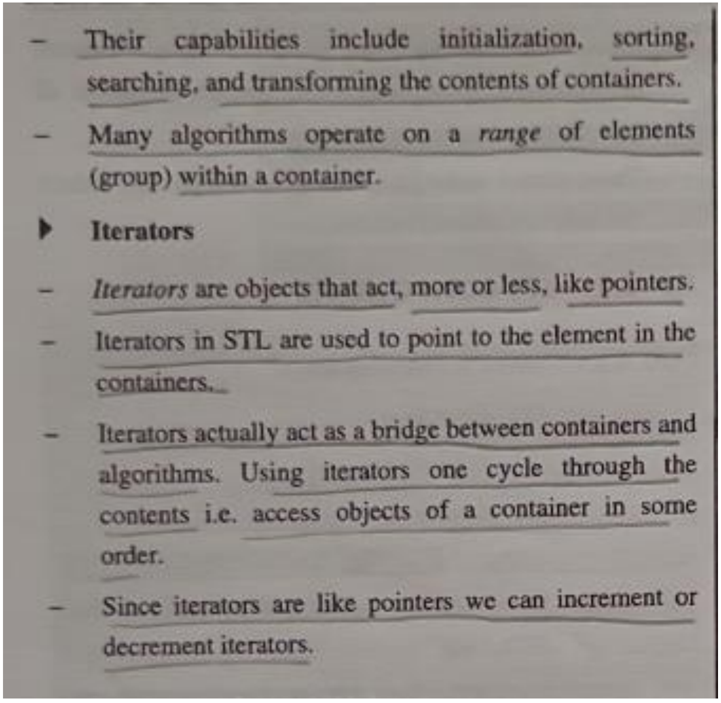
UNIT 6
Standard Template Library (STL)

1) What is purpose of iterator and algorithm



▶ Algorithms

- STL provide number of algorithms that can be used on any container, irrespective of their type.
- Algorithm library contains built in functions which perform complex operations on the data structures.
- Algorithms act on containers. They provide the means by which you will manipulate the contents of containers.

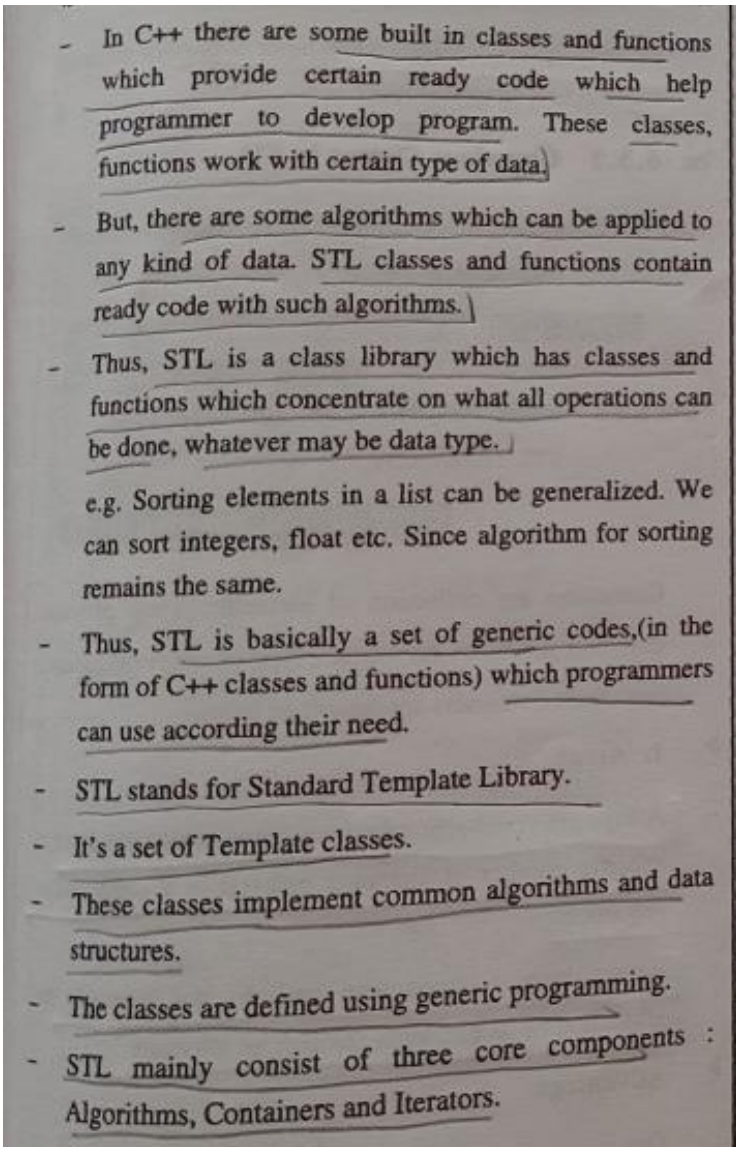


- Their capabilities include initialization, sorting, searching, and transforming the contents of containers.
- Many algorithms operate on a *range* of elements (group) within a container.

▶ Iterators

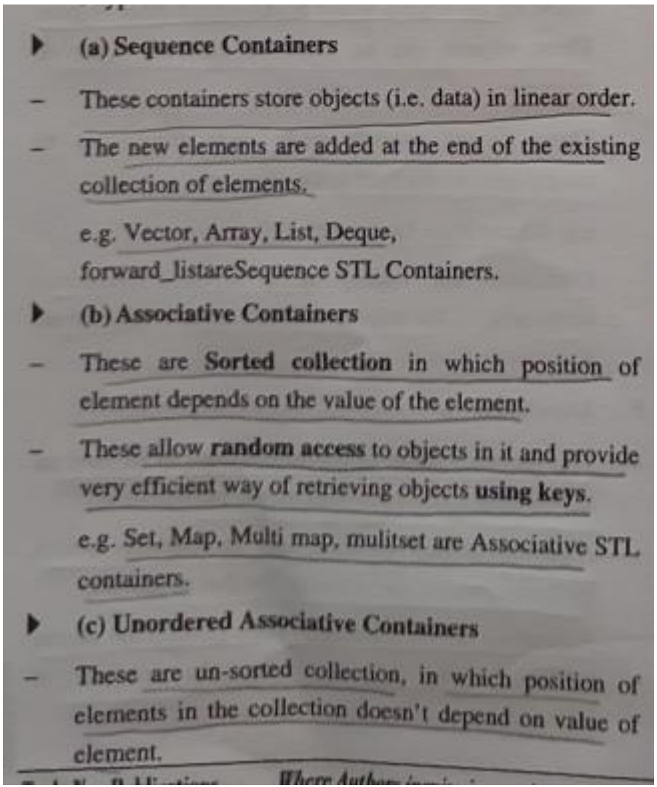
- *Iterators* are objects that act, more or less, like pointers.
- Iterators in STL are used to point to the element in the containers.
- Iterators actually act as a bridge between containers and algorithms. Using iterators one cycle through the contents i.e. access objects of a container in some order.
- Since iterators are like pointers we can increment or decrement iterators.

2) What is STL? List and explain different types of STL containers.



- In C++ there are some built in classes and functions which provide certain ready code which help programmer to develop program. These classes, functions work with certain type of data.
- But, there are some algorithms which can be applied to any kind of data. STL classes and functions contain ready code with such algorithms.
- Thus, STL is a class library which has classes and functions which concentrate on what all operations can be done, whatever may be data type.

  e.g. Sorting elements in a list can be generalized. We can sort integers, float etc. Since algorithm for sorting remains the same.
- Thus, STL is basically a set of generic codes,(in the form of C++ classes and functions) which programmers can use according their need.
- STL stands for Standard Template Library.
- It's a set of Template classes.
- These classes implement common algorithms and data structures.
- The classes are defined using generic programming.
- STL mainly consist of three core components : Algorithms, Containers and Iterators.



- **(a) Sequence Containers**

– These containers store objects (i.e. data) in linear order.

– The new elements are added at the end of the existing collection of elements.

e.g. Vector, Array, List, Deque, forward_listareSequence STL Containers.

- **(b) Associative Containers**

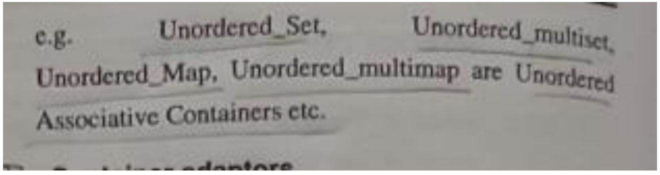
– These are **Sorted collection** in which position of element depends on the value of the element.

– These allow **random access** to objects in it and provide very efficient way of retrieving objects **using keys**.

e.g. Set, Map, Multi map, mulitset are Associative STL containers.

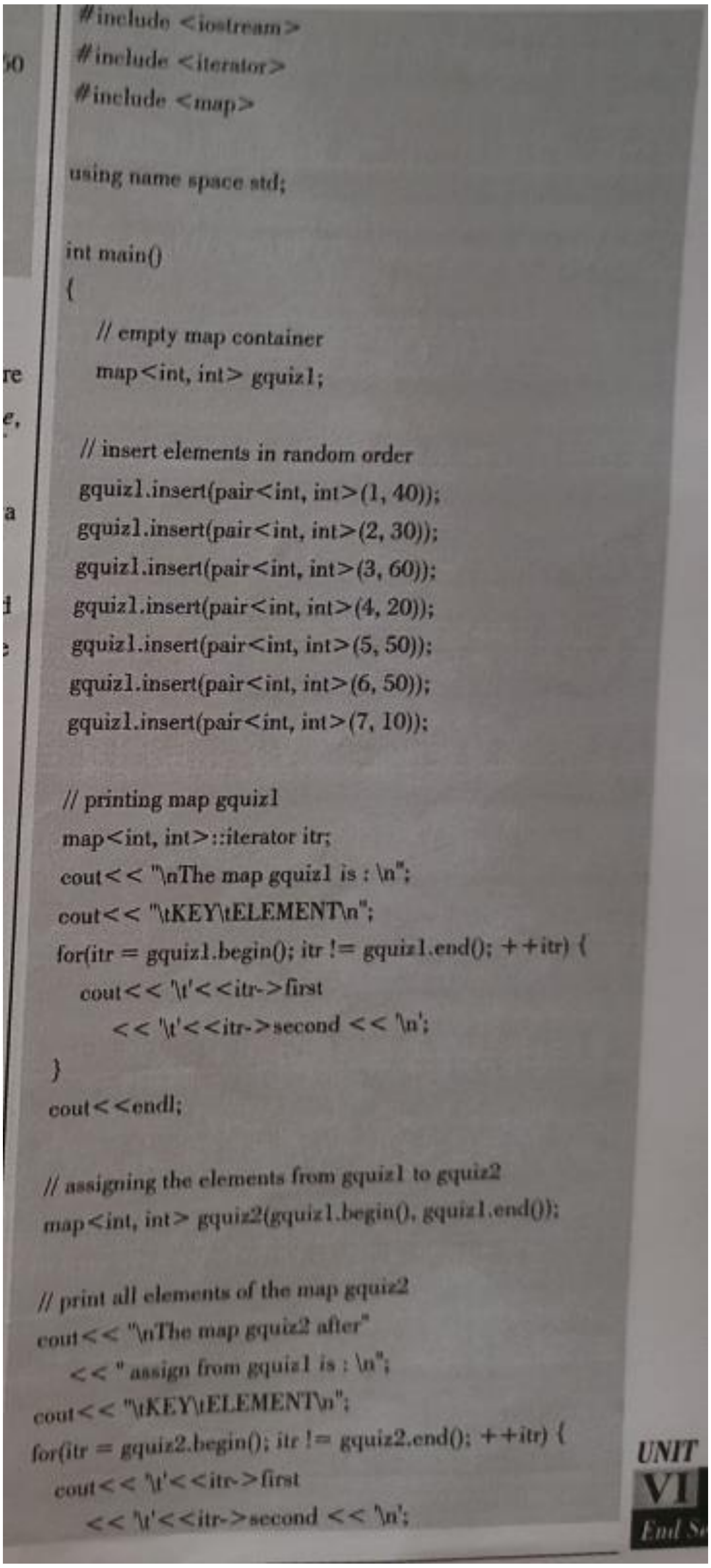
- **(c) Unordered Associative Containers**

– These are un-sorted collection, in which position of elements in the collection doesn't depend on value of element.

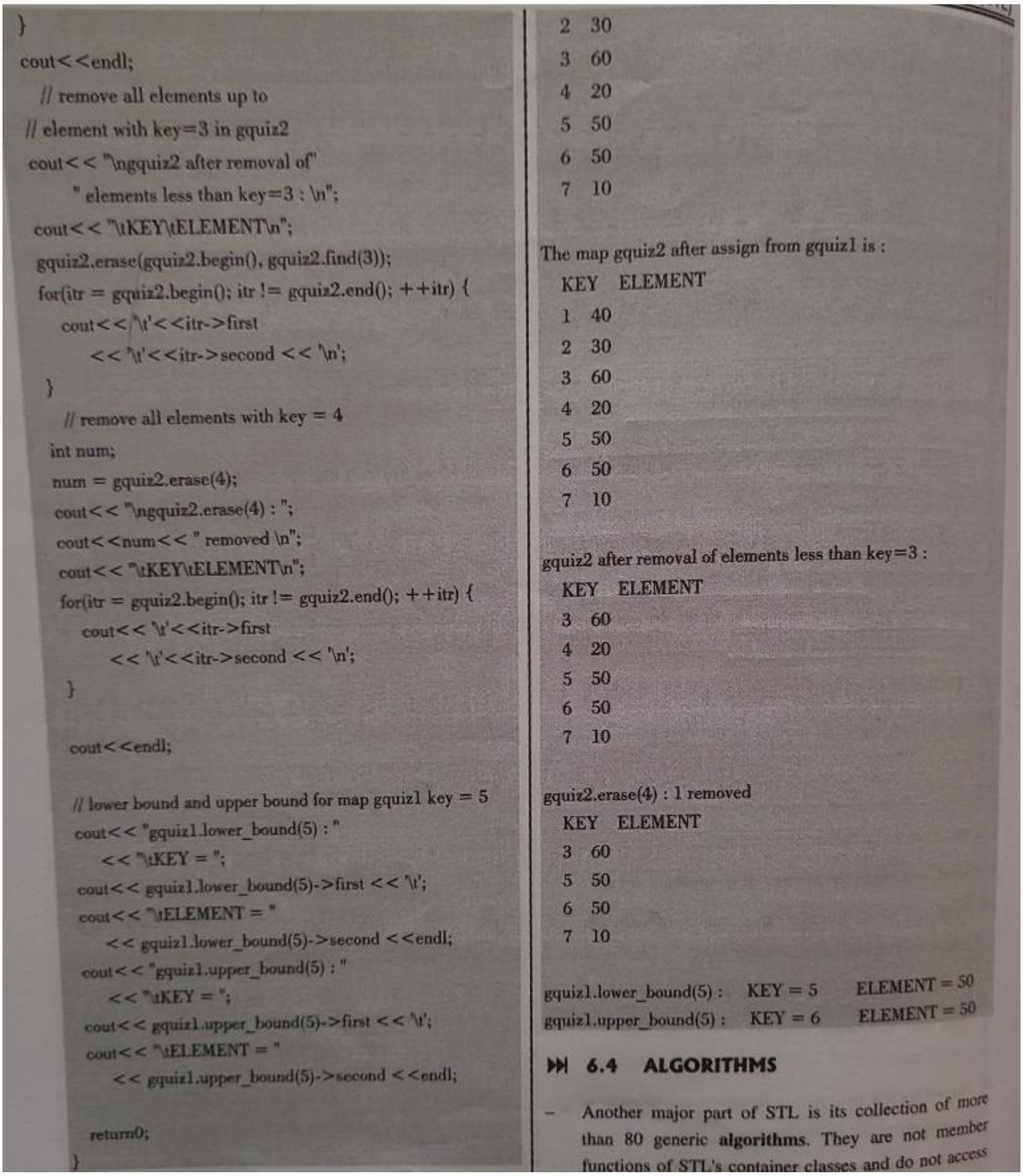


e.g. Unordered_Set, Unordered_multiset, Unordered_Map, Unordered_multimap are Unordered Associative Containers etc.

3) Write a program to implement map in STL



```
#include <iostream>
#include <iterator>
#include <map>

using name space std;

int main()
{
    // empty map container
    map<int, int> gquiz1;

    // insert elements in random order
    gquiz1.insert(pair<int, int>(1, 40));
    gquiz1.insert(pair<int, int>(2, 30));
    gquiz1.insert(pair<int, int>(3, 60));
    gquiz1.insert(pair<int, int>(4, 20));
    gquiz1.insert(pair<int, int>(5, 50));
    gquiz1.insert(pair<int, int>(6, 50));
    gquiz1.insert(pair<int, int>(7, 10));

    // printing map gquiz1
    map<int, int>::iterator itr;
    cout<< "\nThe map gquiz1 is : \n";
    cout<< "\tKEY\tELEMENT\n";
    for(itr = gquiz1.begin(); itr != gquiz1.end(); ++itr) {
        cout<< '\t'<<itr->first
            << '\t'<<itr->second << '\n';
    }
    cout<<endl;

    // assigning the elements from gquiz1 to gquiz2
    map<int, int> gquiz2(gquiz1.begin(), gquiz1.end());

    // print all elements of the map gquiz2
    cout<< "\nThe map gquiz2 after"
        << " assign from gquiz1 is : \n";
    cout<< "\tKEY\tELEMENT\n";
    for(itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
        cout<< '\t'<<itr->first
            << '\t'<<itr->second << '\n';
```



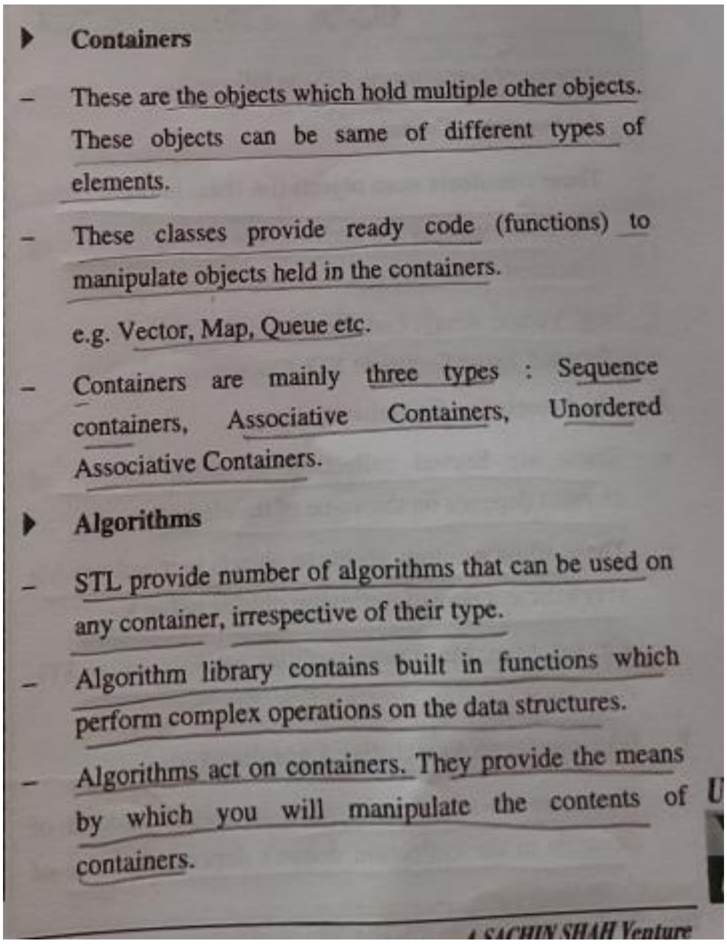
```
}
cout<<endl;
  // remove all elements up to
 // element with key=3 in gquiz2
 cout<< "\ngquiz2 after removal of"
       " elements less than key=3 : \n";
  cout<< "\tKEY\tELEMENT\n";
  gquiz2.erase(gquiz2.begin(), gquiz2.find(3));
  for(itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
     cout<< '\t'<<itr->first
        << '\t'<<itr->second << '\n';
  }
    // remove all elements with key = 4
  int num;
  num = gquiz2.erase(4);
  cout<< "\ngquiz2.erase(4) : ";
  cout<<num<< " removed \n";
  cout<< "\tKEY\tELEMENT\n";
  for(itr = gquiz2.begin(); itr != gquiz2.end(); ++itr) {
     cout<< '\t'<<itr->first
        << '\t'<<itr->second << '\n';
  }

  cout<<endl;

  // lower bound and upper bound for map gquiz1 key = 5
  cout<< "gquiz1.lower_bound(5) : "
     << "\tKEY = ";
  cout<< gquiz1.lower_bound(5)->first << '\t';
  cout<< "\tELEMENT = "
     << gquiz1.lower_bound(5)->second <<endl;
  cout<< "gquiz1.upper_bound(5) : "
     << "\tKEY = ";
  cout<< gquiz1.upper_bound(5)->first << '\t';
  cout<< "\tELEMENT = "
     << gquiz1.upper_bound(5)->second <<endl;

  return0;
}
```

```
  2  30
  3  60
  4  20
  5  50
  6  50
  7  10

The map gquiz2 after assign from gquiz1 is :
  KEY   ELEMENT
  1  40
  2  30
  3  60
  4  20
  5  50
  6  50
  7  10

gquiz2 after removal of elements less than key=3 :
  KEY   ELEMENT
  3  60
  4  20
  5  50
  6  50
  7  10

gquiz2.erase(4) : 1 removed
  KEY   ELEMENT
  3  60
  5  50
  6  50
  7  10

gquiz1.lower_bound(5) :   KEY = 5      ELEMENT = 50
gquiz1.upper_bound(5) :   KEY = 6      ELEMENT = 50
```

⏭ 6.4 ALGORITHMS

- Another major part of STL is its collection of more than 80 generic **algorithms**. They are not member functions of STL's container classes and do not access

4) What are major components of STL



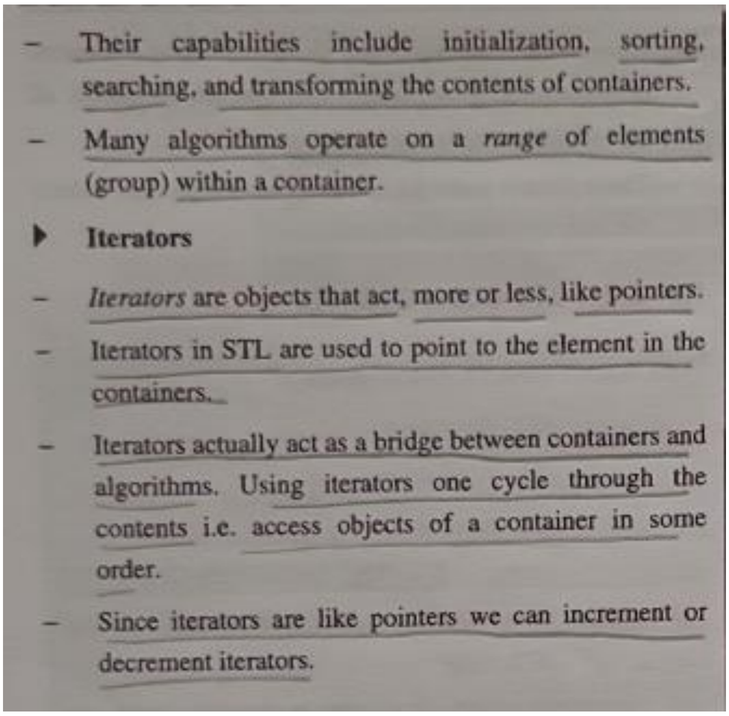
▶ **Containers**

- These are the objects which hold multiple other objects. These objects can be same of different types of elements.
- These classes provide ready code (functions) to manipulate objects held in the containers.

  e.g. Vector, Map, Queue etc.
- Containers are mainly three types : Sequence containers, Associative Containers, Unordered Associative Containers.

▶ **Algorithms**

- STL provide number of algorithms that can be used on any container, irrespective of their type.
- Algorithm library contains built in functions which perform complex operations on the data structures.
- Algorithms act on containers. They provide the means by which you will manipulate the contents of containers.

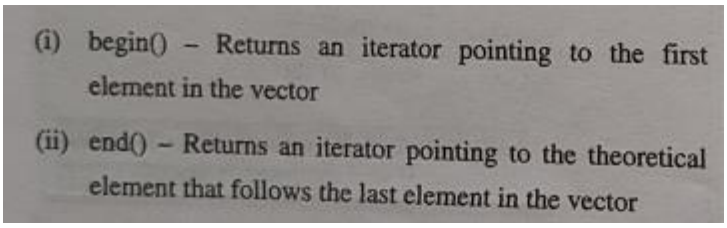


- Their capabilities include initialization, sorting, searching, and transforming the contents of containers.
- Many algorithms operate on a *range* of elements (group) within a container.

▶ **Iterators**

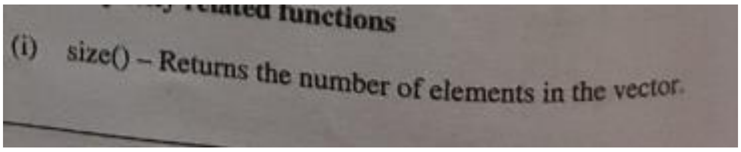
- *Iterators* are objects that act, more or less, like pointers.
- Iterators in STL are used to point to the element in the containers.
- Iterators actually act as a bridge between containers and algorithms. Using iterators one cycle through the contents i.e. access objects of a container in some order.
- Since iterators are like pointers we can increment or decrement iterators.

5) State functions of vector STL. Write a program to explain the same



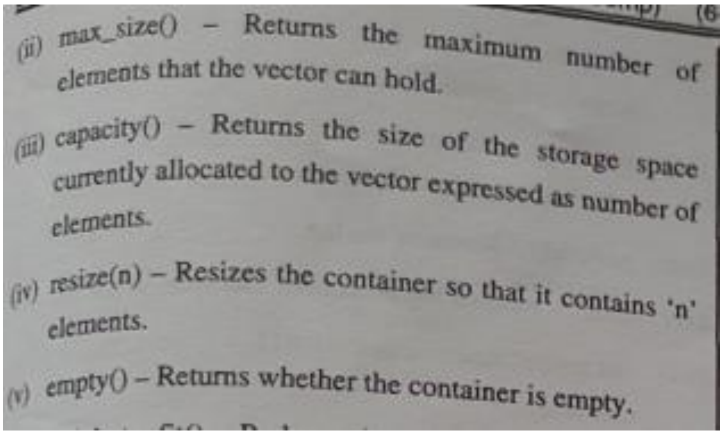
(i) begin() – Returns an iterator pointing to the first element in the vector

(ii) end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector



ated functions

(i) size() – Returns the number of elements in the vector.

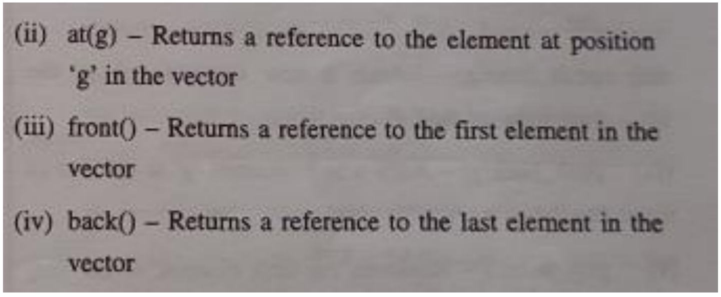


(ii) max_size() – Returns the maximum number of elements that the vector can hold.

(iii) capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.

(iv) resize(n) – Resizes the container so that it contains 'n' elements.

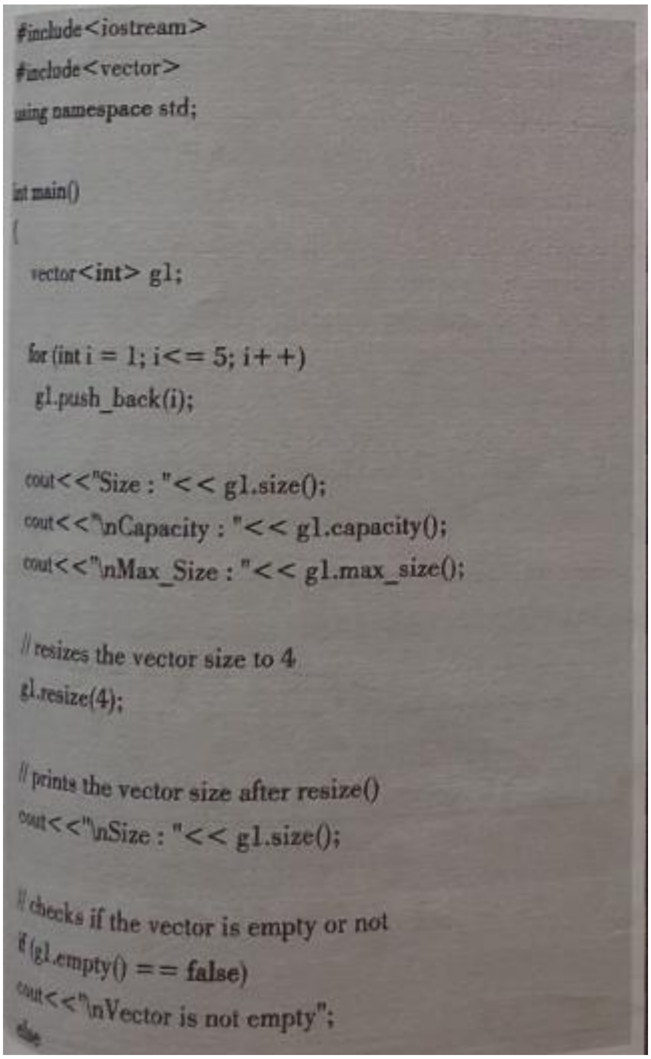
(v) empty() – Returns whether the container is empty.



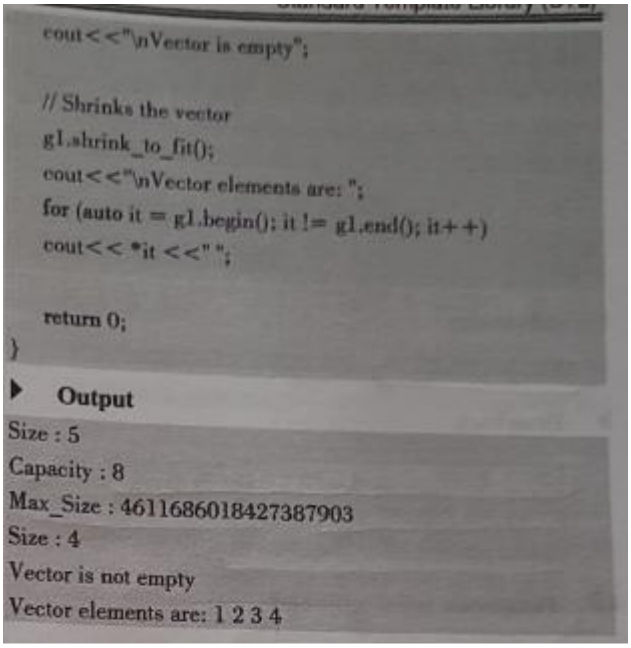
(ii) at(g) – Returns a reference to the element at position 'g' in the vector

(iii) front() – Returns a reference to the first element in the vector

(iv) back() – Returns a reference to the last element in the vector



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

int main()
{
  vector<int> g1;

  for (int i = 1; i<= 5; i++)
   g1.push_back(i);

  cout<<"Size : "<< g1.size();
  cout<<"\nCapacity : "<< g1.capacity();
  cout<<"\nMax_Size : "<< g1.max_size();

  // resizes the vector size to 4
  g1.resize(4);

  // prints the vector size after resize()
  cout<<"\nSize : "<< g1.size();

  // checks if the vector is empty or not
  if (g1.empty() == false)
  cout<<"\nVector is not empty";
  else
```

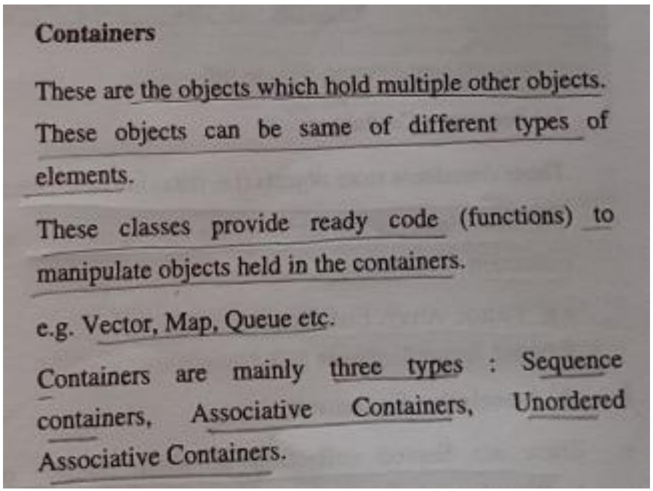


```
  cout<<"\nVector is empty";

  // Shrinks the vector
  g1.shrink_to_fit();
  cout<<"\nVector elements are: ";
  for (auto it = g1.begin(); it != g1.end(); it++)
  cout<< *it <<" ";

  return 0;
}
```

▶ **Output**

```
Size : 5
Capacity : 8
Max_Size : 4611686018427387903
Size : 4
Vector is not empty
Vector elements are: 1 2 3 4
```

6) **What is container? List the container classes in C++. Explain any one of them using program**



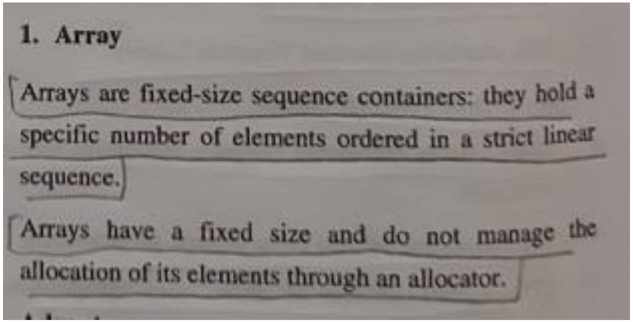
Containers

These are the objects which hold multiple other objects. These objects can be same of different types of elements.

These classes provide ready code (functions) to manipulate objects held in the containers.

e.g. Vector, Map, Queue etc.

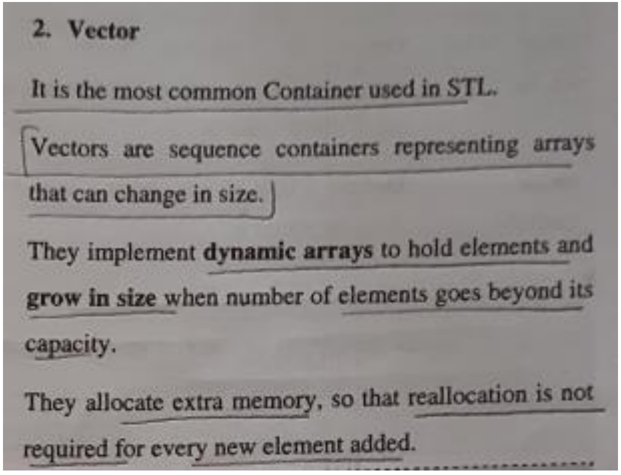
Containers are mainly three types : Sequence containers, Associative Containers, Unordered Associative Containers.



1. Array

Arrays are fixed-size sequence containers: they hold a specific number of elements ordered in a strict linear sequence.

Arrays have a fixed size and do not manage the allocation of its elements through an allocator.



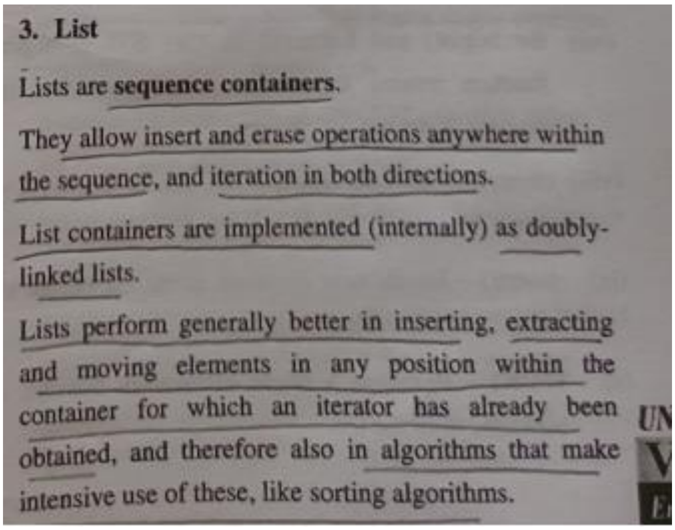
2. Vector

It is the most common Container used in STL.

Vectors are sequence containers representing arrays that can change in size.

They implement **dynamic arrays** to hold elements and **grow in size** when number of elements goes beyond its capacity.

They allocate extra memory, so that reallocation is not required for every new element added.



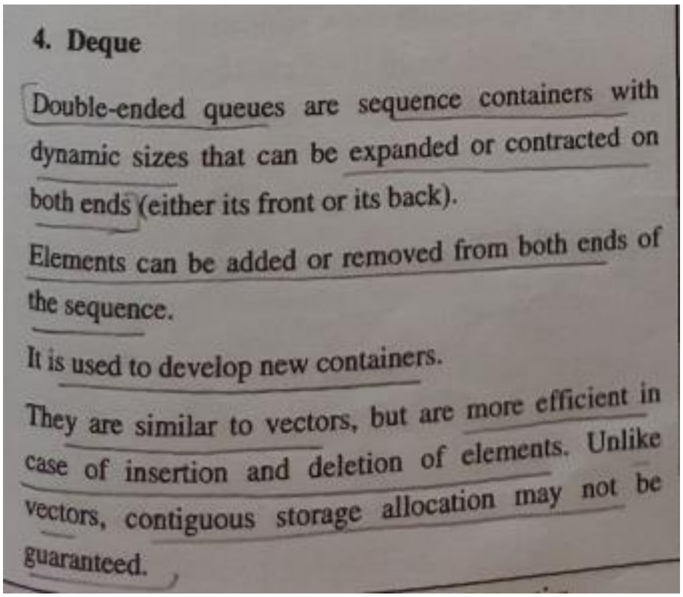
3. List

Lists are **sequence containers**.

They allow insert and erase operations anywhere within the sequence, and iteration in both directions.

List containers are implemented (internally) as doubly-linked lists.

Lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these, like sorting algorithms.



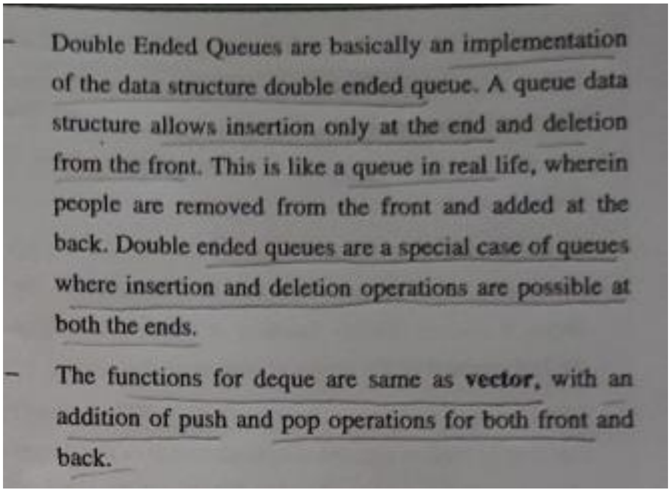
4. Deque

Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Elements can be added or removed from both ends of the sequence.

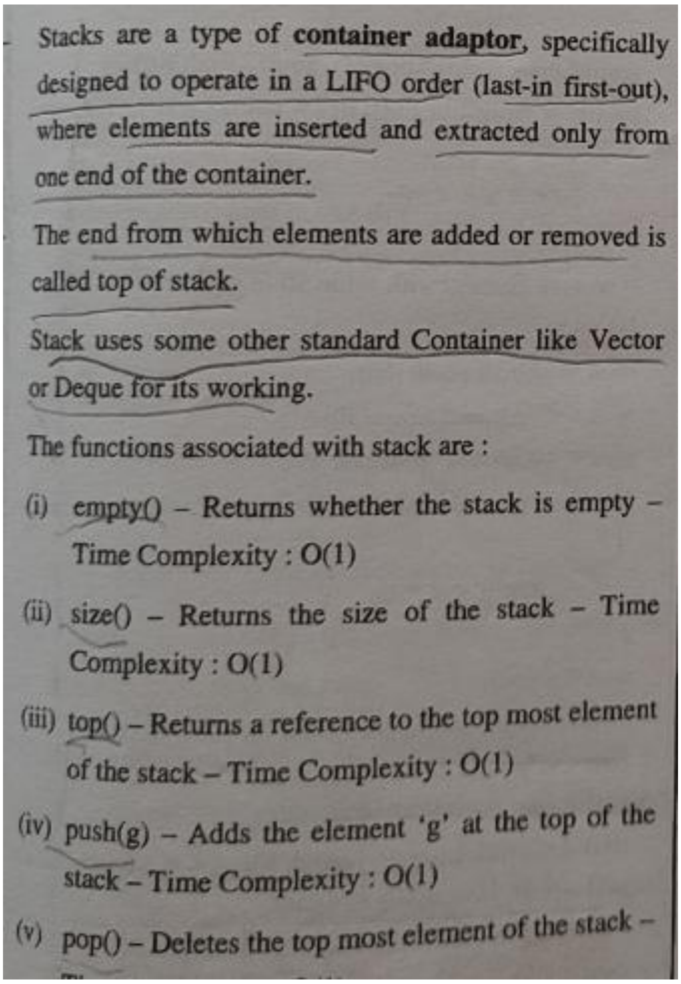
It is used to develop new containers.

They are similar to vectors, but are more efficient in case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed.



- Double Ended Queues are basically an implementation of the data structure double ended queue. A queue data structure allows insertion only at the end and deletion from the front. This is like a queue in real life, wherein people are removed from the front and added at the back. Double ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.
- The functions for deque are same as **vector**, with an addition of push and pop operations for both front and back.

5. Stack



- Stacks are a type of **container adaptor**, specifically designed to operate in a LIFO order (last-in first-out), where elements are inserted and extracted only from one end of the container.
- The end from which elements are added or removed is called top of stack.

Stack uses some other standard Container like Vector or Deque for its working.

The functions associated with stack are :

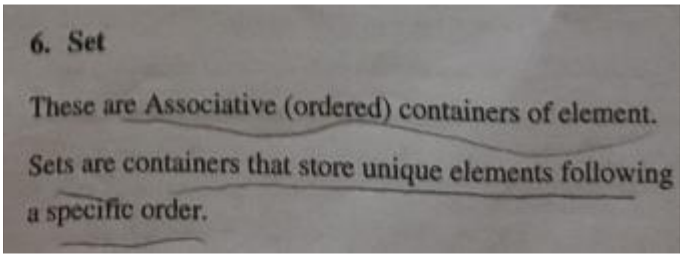
(i) empty() – Returns whether the stack is empty – Time Complexity : O(1)

(ii) size() – Returns the size of the stack – Time Complexity : O(1)

(iii) top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)

(iv) push(g) – Adds the element 'g' at the top of the stack – Time Complexity : O(1)

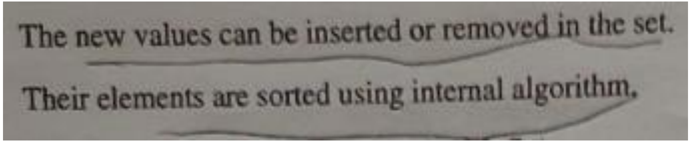
(v) pop() – Deletes the top most element of the stack –



6. Set

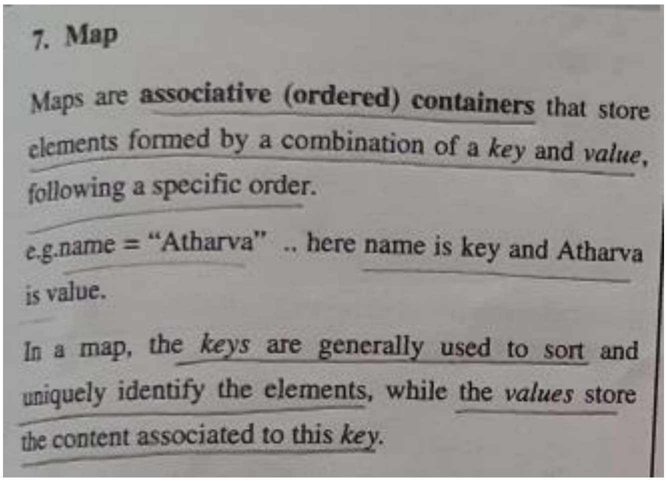
These are Associative (ordered) containers of element.

Sets are containers that store unique elements following a specific order.



The new values can be inserted or removed in the set.

Their elements are sorted using internal algorithm.



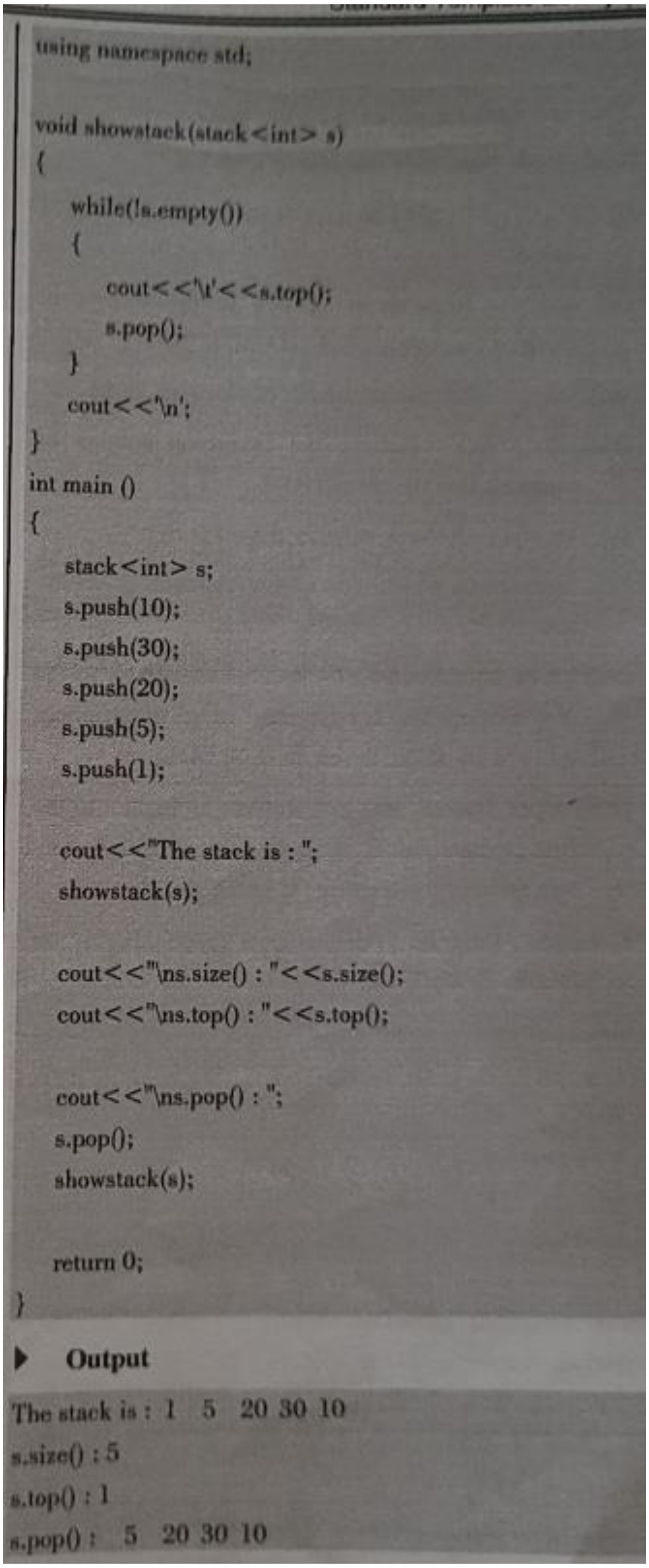
7. Map

Maps are **associative (ordered) containers** that store elements formed by a combination of a *key* and *value*, following a specific order.

e.g.name = "Atharva" .. here name is key and Atharva is value.

In a map, the *keys* are generally used to sort and uniquely identify the elements, while the *values* store the content associated to this *key*.

```
container
#include<stack>
#include<iostream>
```

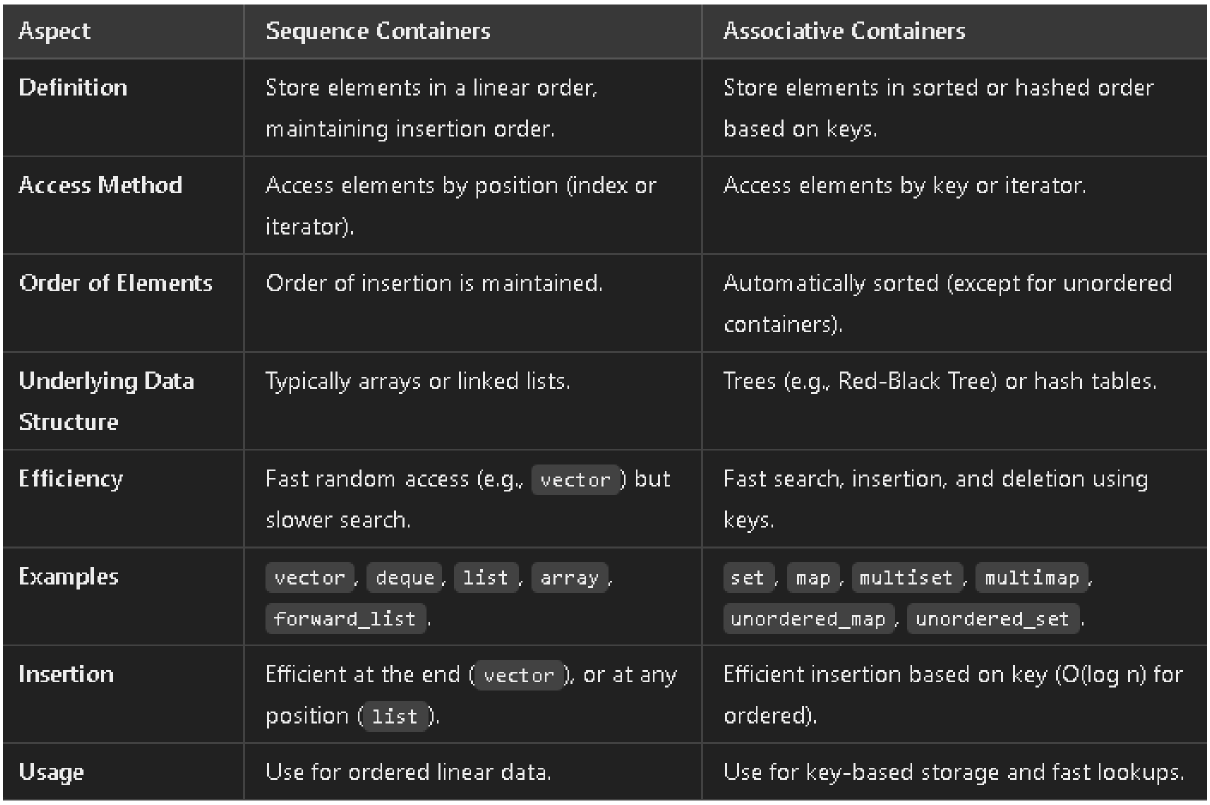


```
using namespace std;

void showstack(stack<int> s)
{
    while(!s.empty())
    {
        cout<<'\t'<<s.top();
        s.pop();
    }
    cout<<'\n';
}
int main ()
{
    stack<int> s;
    s.push(10);
    s.push(30);
    s.push(20);
    s.push(5);
    s.push(1);

    cout<<"The stack is : ";
    showstack(s);

    cout<<"\ns.size() : "<<s.size();
    cout<<"\ns.top() : "<<s.top();

    cout<<"\ns.pop() : ";
    s.pop();
    showstack(s);

    return 0;
}
```

Output

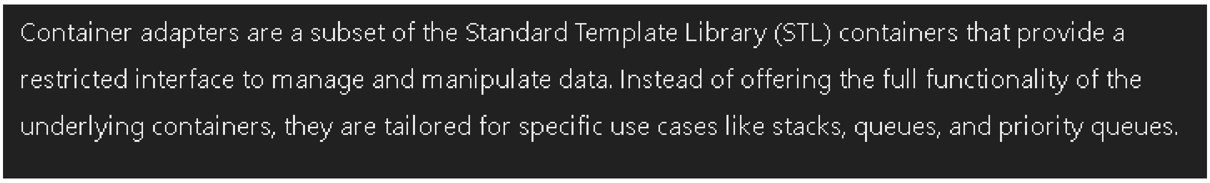
```
The stack is :  1   5   20  30  10
s.size() : 5
s.top() : 1
s.pop() :   5   20  30  10
```

7) Differentiate between sequence containers and associative containers in the STL. Provide examples of each

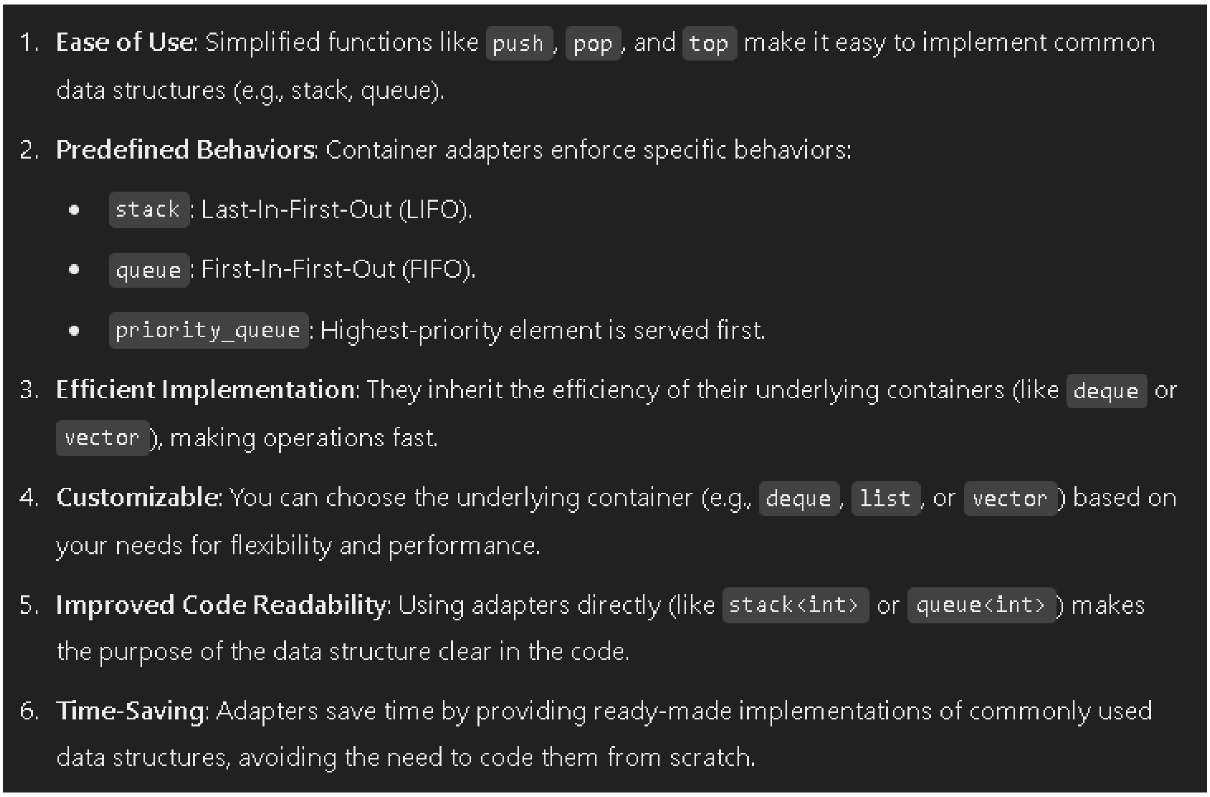


Aspect	Sequence Containers	Associative Containers
Definition	Store elements in a linear order, maintaining insertion order.	Store elements in sorted or hashed order based on keys.
Access Method	Access elements by position (index or iterator).	Access elements by key or iterator.
Order of Elements	Order of insertion is maintained.	Automatically sorted (except for unordered containers).
Underlying Data Structure	Typically arrays or linked lists.	Trees (e.g., Red-Black Tree) or hash tables.
Efficiency	Fast random access (e.g., `vector`) but slower search.	Fast search, insertion, and deletion using keys.
Examples	`vector`, `deque`, `list`, `array`, `forward_list`.	`set`, `map`, `multiset`, `multimap`, `unordered_map`, `unordered_set`.
Insertion	Efficient at the end (`vector`), or at any position (`list`).	Efficient insertion based on key (O(log n) for ordered).
Usage	Use for ordered linear data.	Use for key-based storage and fast lookups.

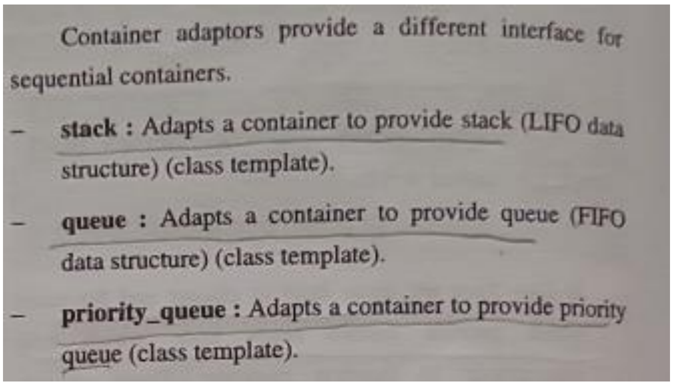
8) Discuss the advantages of using container adapters in the STL. Provide examples of container adapters



Container adapters are a subset of the Standard Template Library (STL) containers that provide a restricted interface to manage and manipulate data. Instead of offering the full functionality of the underlying containers, they are tailored for specific use cases like stacks, queues, and priority queues.



1. **Ease of Use**: Simplified functions like `push`, `pop`, and `top` make it easy to implement common data structures (e.g., stack, queue).
2. **Predefined Behaviors**: Container adapters enforce specific behaviors:
   - `stack`: Last-In-First-Out (LIFO).
   - `queue`: First-In-First-Out (FIFO).
   - `priority_queue`: Highest-priority element is served first.
3. **Efficient Implementation**: They inherit the efficiency of their underlying containers (like `deque` or `vector`), making operations fast.
4. **Customizable**: You can choose the underlying container (e.g., `deque`, `list`, or `vector`) based on your needs for flexibility and performance.
5. **Improved Code Readability**: Using adapters directly (like `stack<int>` or `queue<int>`) makes the purpose of the data structure clear in the code.
6. **Time-Saving**: Adapters save time by providing ready-made implementations of commonly used data structures, avoiding the need to code them from scratch.



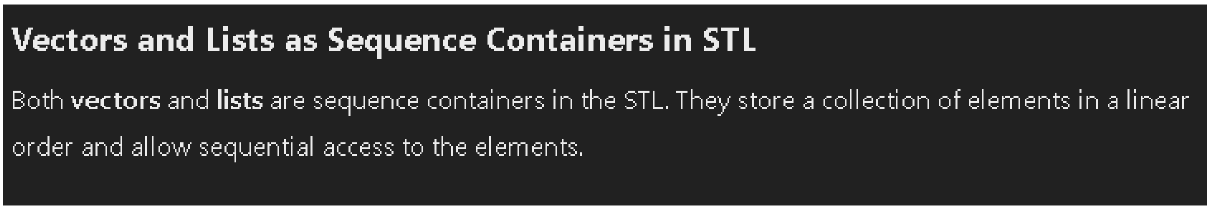
Container adaptors provide a different interface for sequential containers.

– **stack :** Adapts a container to provide stack (LIFO data structure) (class template).

– **queue :** Adapts a container to provide queue (FIFO data structure) (class template).

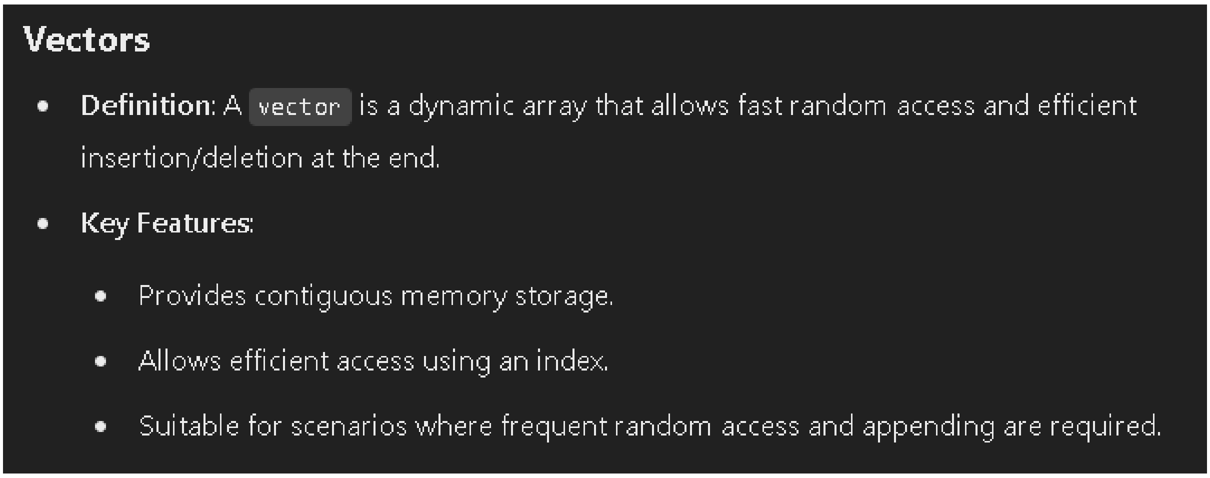
– **priority_queue :** Adapts a container to provide priority queue (class template).

9) **How can vectors and lists be used as sequence containers in the STL? Explain with a appropriate example**



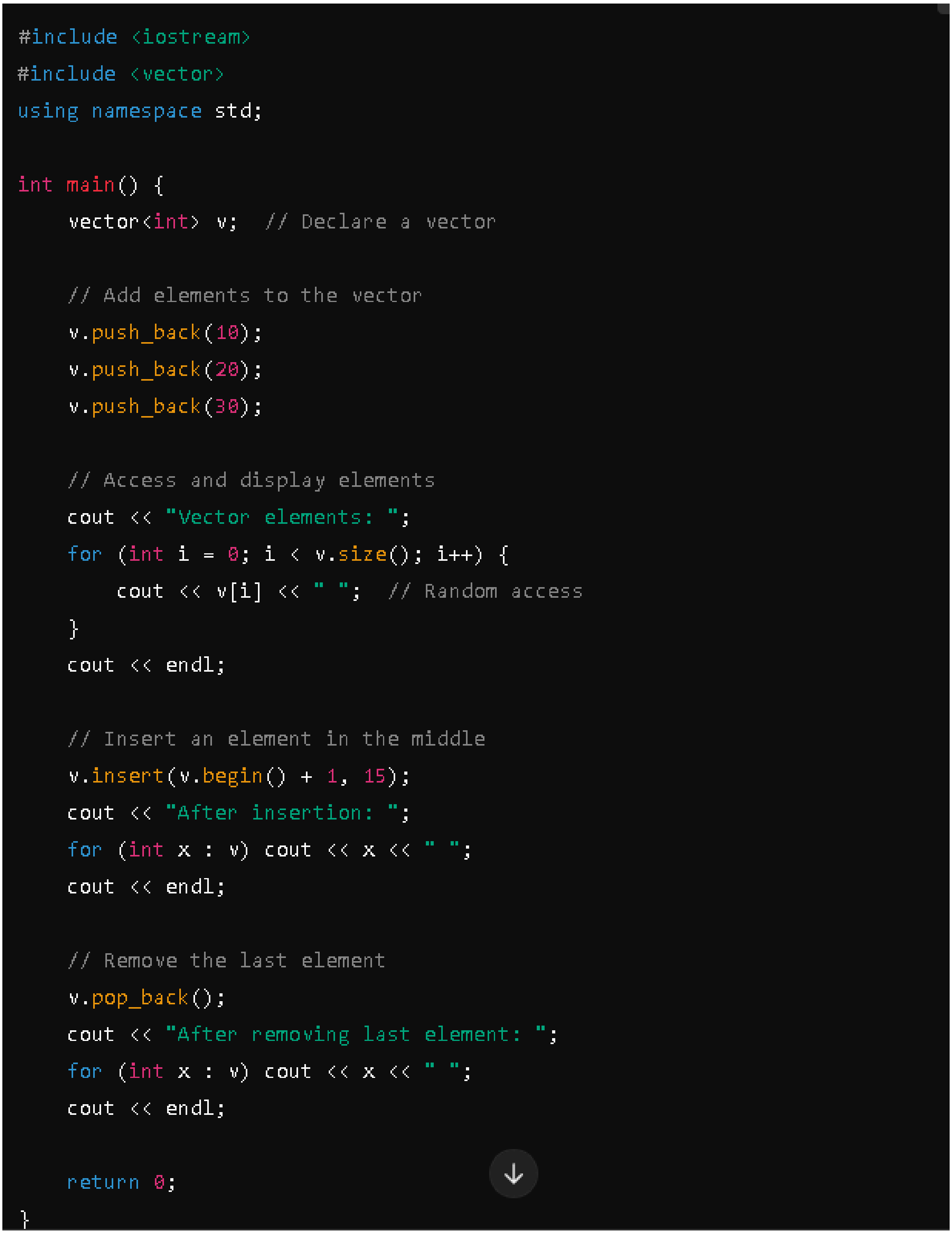
Vectors and Lists as Sequence Containers in STL

Both **vectors** and **lists** are sequence containers in the STL. They store a collection of elements in a linear order and allow sequential access to the elements.

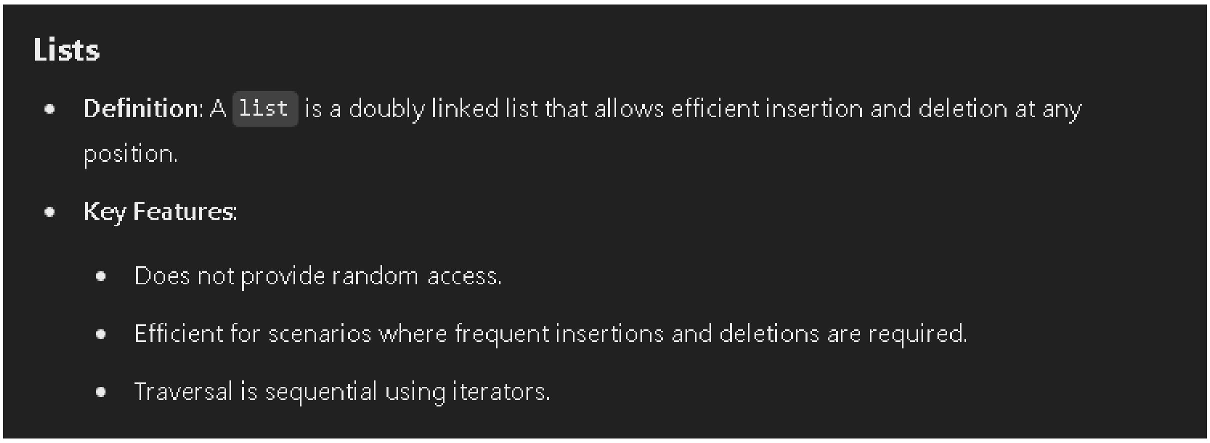


Vectors

- **Definition**: A `vector` is a dynamic array that allows fast random access and efficient insertion/deletion at the end.
- **Key Features**:
  - Provides contiguous memory storage.
  - Allows efficient access using an index.
  - Suitable for scenarios where frequent random access and appending are required.

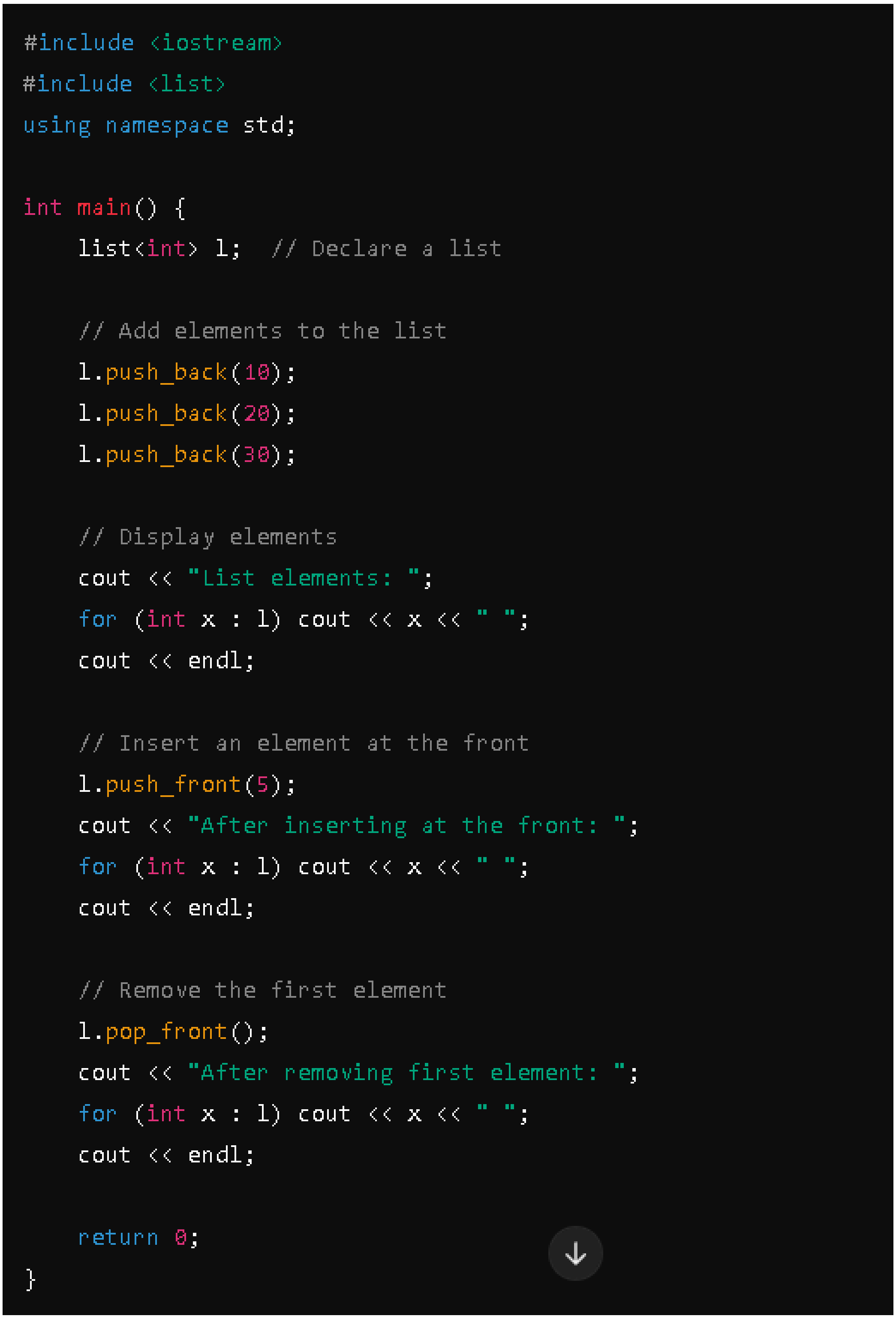


```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v;  // Declare a vector

    // Add elements to the vector
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);

    // Access and display elements
    cout << "Vector elements: ";
    for (int i = 0; i < v.size(); i++) {
        cout << v[i] << " ";  // Random access
    }
    cout << endl;

    // Insert an element in the middle
    v.insert(v.begin() + 1, 15);
    cout << "After insertion: ";
    for (int x : v) cout << x << " ";
    cout << endl;

    // Remove the last element
    v.pop_back();
    cout << "After removing last element: ";
    for (int x : v) cout << x << " ";
    cout << endl;

    return 0;
}
```



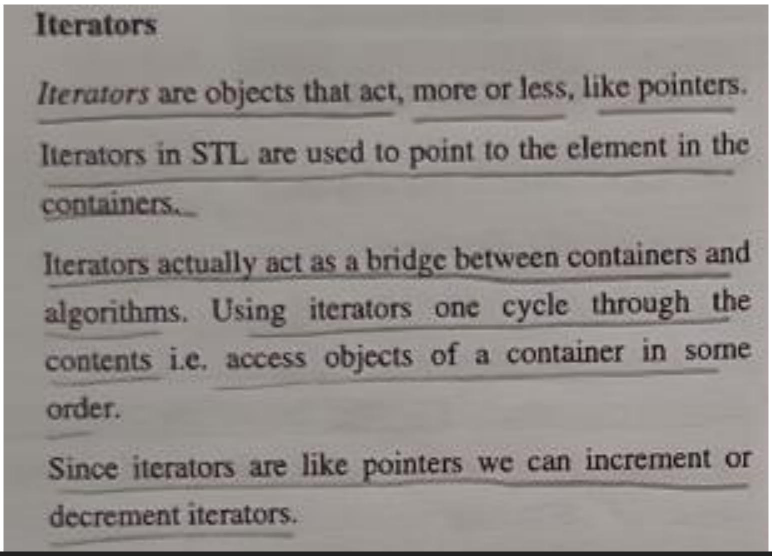
Lists

- **Definition:** A `list` is a doubly linked list that allows efficient insertion and deletion at any position.
- **Key Features:**
  - Does not provide random access.
  - Efficient for scenarios where frequent insertions and deletions are required.
  - Traversal is sequential using iterators.



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

int main() {
    list<int> l;  // Declare a list

    // Add elements to the list
    l.push_back(10);
    l.push_back(20);
    l.push_back(30);

    // Display elements
    cout << "List elements: ";
    for (int x : l) cout << x << " ";
    cout << endl;

    // Insert an element at the front
    l.push_front(5);
    cout << "After inserting at the front: ";
    for (int x : l) cout << x << " ";
    cout << endl;

    // Remove the first element
    l.pop_front();
    cout << "After removing first element: ";
    for (int x : l) cout << x << " ";
    cout << endl;

    return 0;
}
```

10) Explain the concept of iterators in the STL. Differentiate between iterator and pointers.



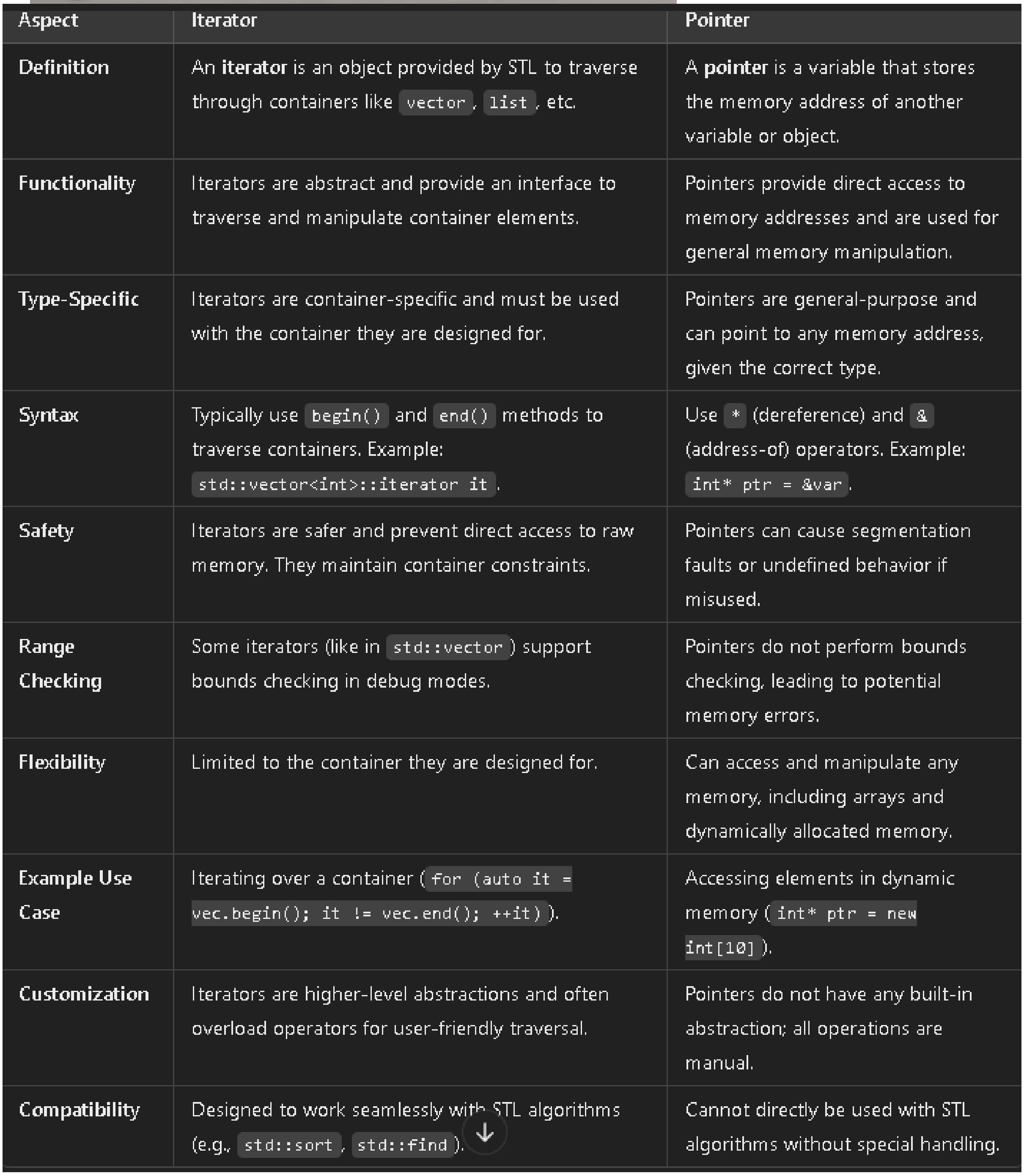
Iterators

Iterators are objects that act, more or less, like pointers.

Iterators in STL are used to point to the element in the containers.

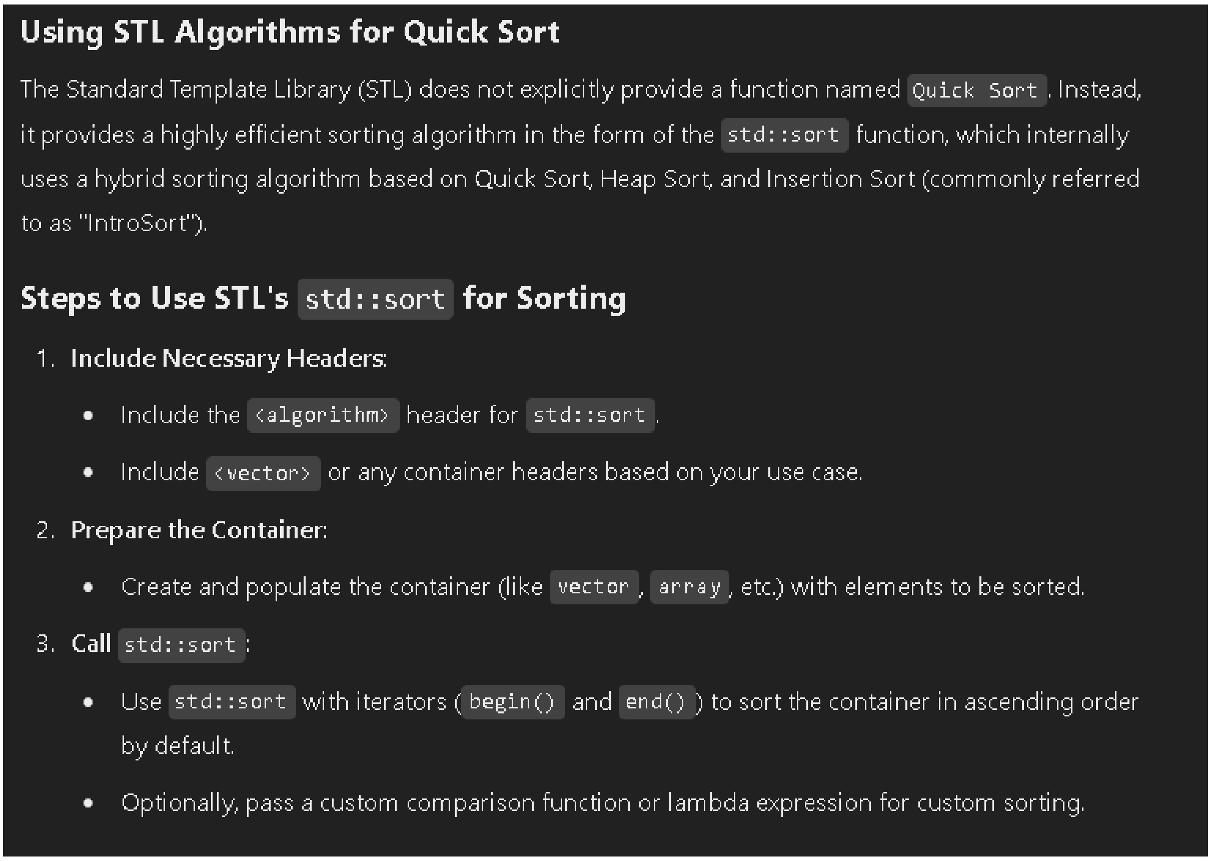
Iterators actually act as a bridge between containers and algorithms. Using iterators one cycle through the contents i.e. access objects of a container in some order.

Since iterators are like pointers we can increment or decrement iterators.



Aspect	Iterator	Pointer
Definition	An **iterator** is an object provided by STL to traverse through containers like `vector`, `list`, etc.	A **pointer** is a variable that stores the memory address of another variable or object.
Functionality	Iterators are abstract and provide an interface to traverse and manipulate container elements.	Pointers provide direct access to memory addresses and are used for general memory manipulation.
Type-Specific	Iterators are container-specific and must be used with the container they are designed for.	Pointers are general-purpose and can point to any memory address, given the correct type.
Syntax	Typically use `begin()` and `end()` methods to traverse containers. Example: `std::vector<int>::iterator it`.	Use `*` (dereference) and `&` (address-of) operators. Example: `int* ptr = &var`.
Safety	Iterators are safer and prevent direct access to raw memory. They maintain container constraints.	Pointers can cause segmentation faults or undefined behavior if misused.
Range Checking	Some iterators (like in `std::vector`) support bounds checking in debug modes.	Pointers do not perform bounds checking, leading to potential memory errors.
Flexibility	Limited to the container they are designed for.	Can access and manipulate any memory, including arrays and dynamically allocated memory.
Example Use Case	Iterating over a container (`for (auto it = vec.begin(); it != vec.end(); ++it)`).	Accessing elements in dynamic memory (`int* ptr = new int[10]`).
Customization	Iterators are higher-level abstractions and often overload operators for user-friendly traversal.	Pointers do not have any built-in abstraction; all operations are manual.
Compatibility	Designed to work seamlessly with STL algorithms (e.g., `std::sort`, `std::find`).	Cannot directly be used with STL algorithms without special handling.

11) Describe the process of using the STL algorithms for Quick sort.

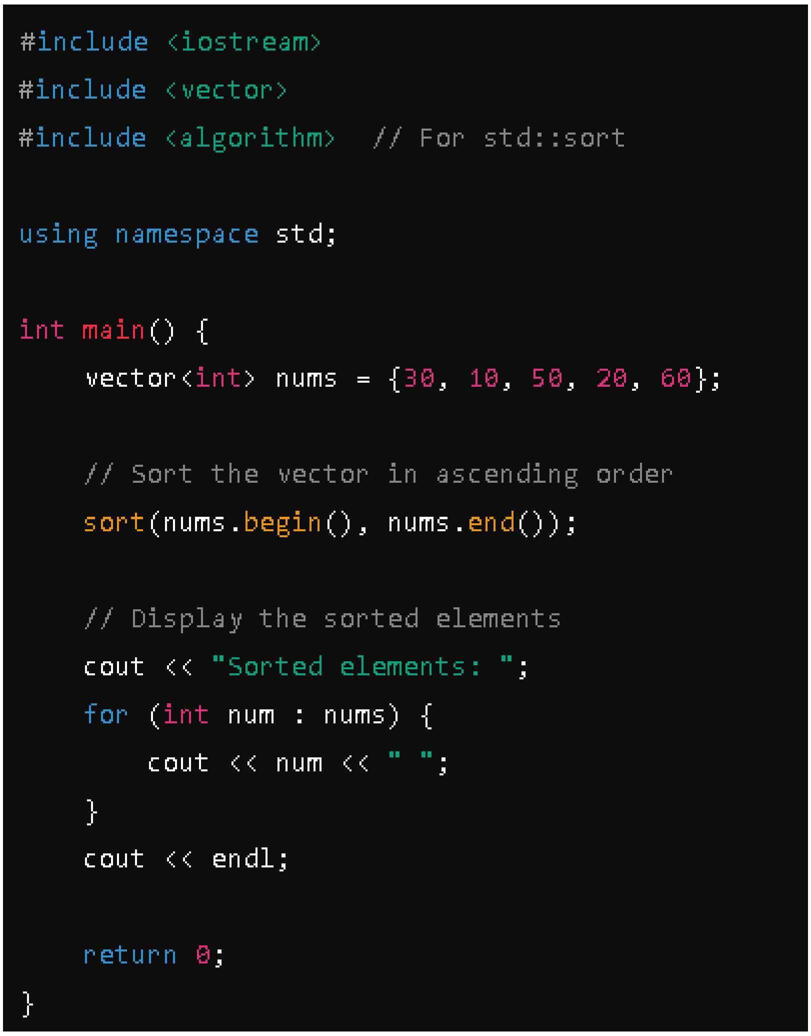


Using STL Algorithms for Quick Sort

The Standard Template Library (STL) does not explicitly provide a function named `Quick Sort`. Instead, it provides a highly efficient sorting algorithm in the form of the `std::sort` function, which internally uses a hybrid sorting algorithm based on Quick Sort, Heap Sort, and Insertion Sort (commonly referred to as "IntroSort").

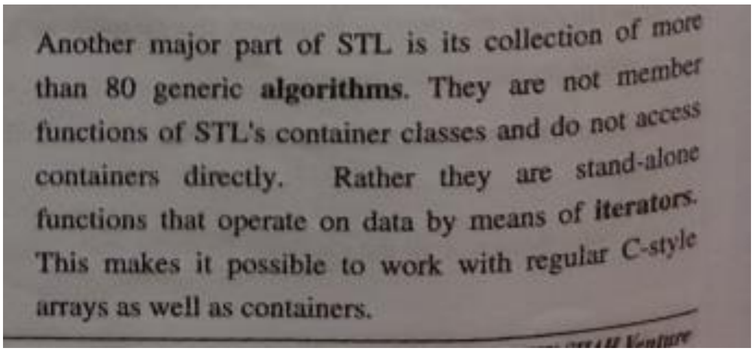
Steps to Use STL's `std::sort` for Sorting

1. **Include Necessary Headers:**
   - Include the `<algorithm>` header for `std::sort`.
   - Include `<vector>` or any container headers based on your use case.
2. **Prepare the Container:**
   - Create and populate the container (like `vector`, `array`, etc.) with elements to be sorted.
3. **Call `std::sort`:**
   - Use `std::sort` with iterators (`begin()` and `end()`) to sort the container in ascending order by default.
   - Optionally, pass a custom comparison function or lambda expression for custom sorting.

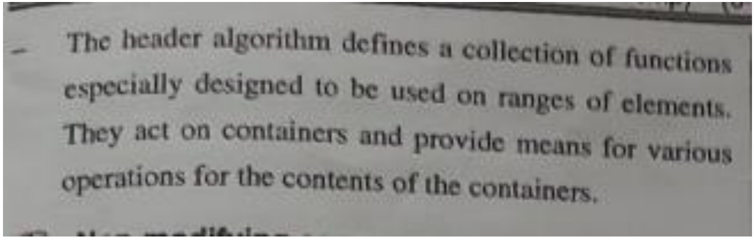


```
#include <iostream>
#include <vector>
#include <algorithm>  // For std::sort

using namespace std;

int main() {
    vector<int> nums = {30, 10, 50, 20, 60};

    // Sort the vector in ascending order
    sort(nums.begin(), nums.end());

    // Display the sorted elements
    cout << "Sorted elements: ";
    for (int num : nums) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

12) What is an algorithm in STL? Enlist algorithms and explain any algorithm in detail



Another major part of STL is its collection of more than 80 generic **algorithms**. They are not member functions of STL's container classes and do not access containers directly. Rather they are stand-alone functions that operate on data by means of **iterators**. This makes it possible to work with regular C-style arrays as well as containers.



- The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

- The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

☞ Non-modifying sequence operations

- std :: all_of : Test condition on all elements in range
- std :: any_of : Test if any element in range fulfils condition
- std :: none_of : Test if no elements fulfil condition
- std :: for_each : Apply function to range
- std :: find : Find value in range
- std :: find_if : Find element in range
- std :: find_if_not : Find element in range (negative condition)
- std :: find_end : Find last subsequence in range
- std :: find_first_of : Find element from set in range
- std :: adjacent_find : Find equal adjacent elements in range
- std :: count : Count appearances of value in range
- std :: count_if : Return number of elements in range satisfying condition
- std :: mismatch : Return first position where two ranges differ
- std::equal : Test whether the elements in two ranges are equal
- std :: is_permutation : Test whether range is permutation of another
- std :: search : Search range for subsequence
- std ::search_n : Search range for element

☞ Modifying sequence operations

- std :: copy : Copy range of elements
- std :: copy_n : Copy elements
- std :: copy_if : Copy certain elements of range
- std :: copy_backward : Copy range of elements backward
- std::move : Move range of elements
- std :: move_backward : Move range of elements backward
- std :: swap : Exchange values of two objects
- std ::swap_ranges : Exchange values of two ranges
- std :: iter_swap : Exchange values of objects pointed to by two iterators
- std ::transform : Transform range
- std ::replace : Replace value in range
- std ::replace_if : Replace values in range
- std :: replace_copy : Copy range replacing value
- std :: replace_copy_if : Copy range replacing value
- std ::fill : Fill range with value
- std :: fill_n : Fill sequence with value
- std ::generate : Generate values for range with function
- std ::generate_n : Generate values for sequence with function
- std ::remove : Remove value from range
- std :: remove_if : Remove elements from range
- remove_copy : Copy range removing value
- remove_copy_if : Copy range removing values
- std ::unique : Remove consecutive duplicates in range
- std :: unique_copy : Copy range removing duplicates
- std ::reverse : Reverse range
- std :: reverse_copy : Copy range reversed
- std :: rotate : Rotate left the elements in range
- std :: rotate_copy : Copy range rotated left
- std :: random_shuffle : Randomly rearrange elements in range
- std :: shuffle : Randomly rearrange elements in range using generator

☞ Partition Operations

- std :: is_partitioned : Test whether range is partitioned
- std :: partition : Partition range in two
- std :: stable_partition : Partition range in two – stable ordering
- partition_copy : Partition range into two
- partition_point : Get partition point

☞ Sorting

- std :: sort : Sort elements in range
- std :: stable_sort : Sort elements preserving order of equivalents
- std :: partial_sort : Partially sort elements in range
- std :: partial_sort_copy : Copy and partially sort range
- std :: is_sorted : Check whether range is sorted
- std :: is_sorted_until : Find first unsorted element in range
- std :: nth_element : Sort element in range

☞ Binary search (operating on partitioned/sorted ranges)

- std :: lower_bound : Return iterator to lower bound
- std :: upper_bound : Return iterator to upper bound
- std :: equal_range : Get subrange of equal elements
- std :: binary_search : Test if value exists in sorted sequence

☞ Merge (operating on sorted ranges)

- std :: merge : Merge sorted ranges
- std :: inplace_merge : Merge consecutive sorted ranges
- std :: includes : Test whether sorted range includes another sorted range
- std :: set_union : Union of two sorted ranges
- std :: set_intersection : Intersection of two sorted ranges
- std :: set_difference : Difference of two sorted ranges
- std :: set_symmetric_difference : Symmetric difference of two sorted ranges

☞ Heap Operations

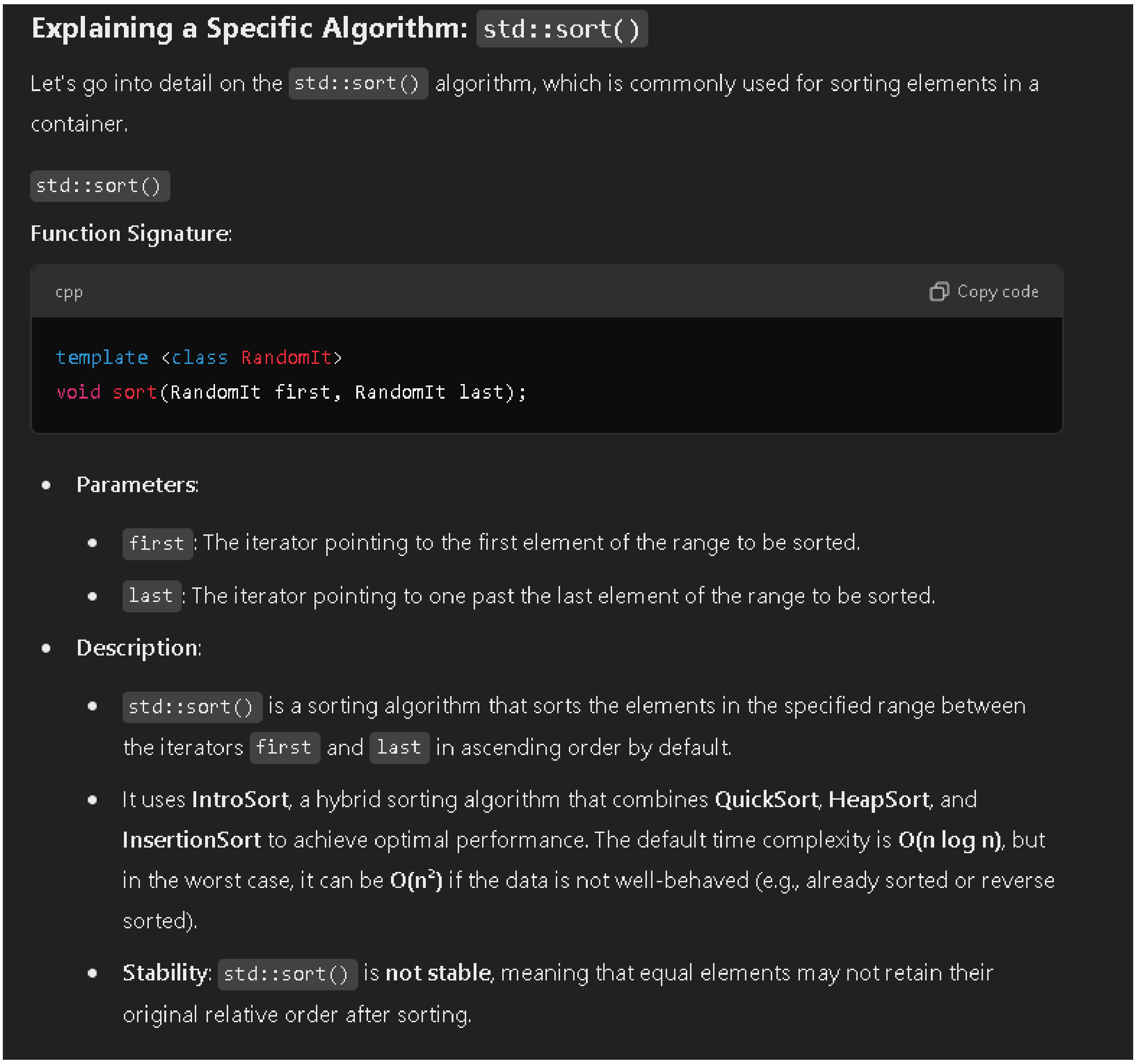
- std :: push_heap : Push element into heap range
- std :: pop_heap : Pop element from heap range
- std :: make_heap : Make heap from range
- std :: sort_heap : Sort elements of heap
- std :: is_heap : Test if range is heap
- std :: is_heap_until : Find first element not in heap order
- std :: max : Return the largest
- std :: minmax : Return smallest and largest elements
- std :: min_element : Return smallest element in range
- std :: max_element : Return largest element in range
- std :: minmax_element : Return smallest and largest elements in range

☞ Other Operations

- std :: lexicographical_compare : Lexicographical less-than comparison
- std :: next_permutation : Transform range to next permutation
- std :: prev_permutation : Transform range to previous permutation

6.4.1 Sort

- Sorting is one of the most basic functions applied to data. It means arranging the data in a particular fashion, which can be increasing or decreasing. There is a built in function in C++ STL by the name of sort().
- This function internally uses IntroSort. In more details it is implemented using hybrid of QuickSort, HeapSort and InsertionSort. By default, it uses QuickSort but if QuickSort is doing unfair partitioning and taking more than N*logN time, it switches to HeapSort and when the array size becomes really small, it switches to InsertionSort.



Explaining a Specific Algorithm: `std::sort()`

Let's go into detail on the `std::sort()` algorithm, which is commonly used for sorting elements in a container.

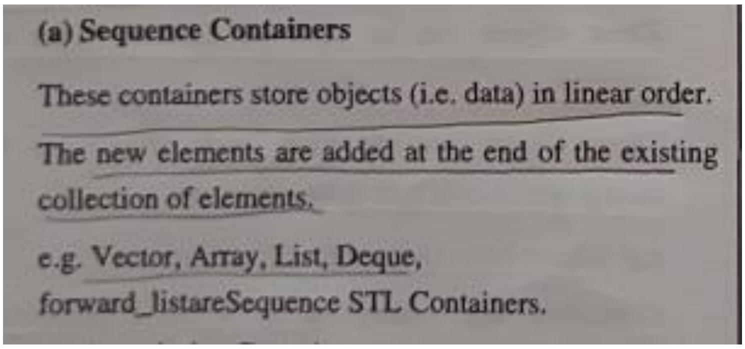
`std::sort()`

Function Signature:

```cpp
template <class RandomIt>
void sort(RandomIt first, RandomIt last);
```

- **Parameters:**
  - `first`: The iterator pointing to the first element of the range to be sorted.
  - `last`: The iterator pointing to one past the last element of the range to be sorted.
- **Description:**
  - `std::sort()` is a sorting algorithm that sorts the elements in the specified range between the iterators `first` and `last` in ascending order by default.
  - It uses **IntroSort**, a hybrid sorting algorithm that combines **QuickSort**, **HeapSort**, and **InsertionSort** to achieve optimal performance. The default time complexity is **O(n log n)**, but in the worst case, it can be **O(n^2)** if the data is not well-behaved (e.g., already sorted or reverse sorted).
  - **Stability**: `std::sort()` is **not stable**, meaning that equal elements may not retain their original relative order after sorting.

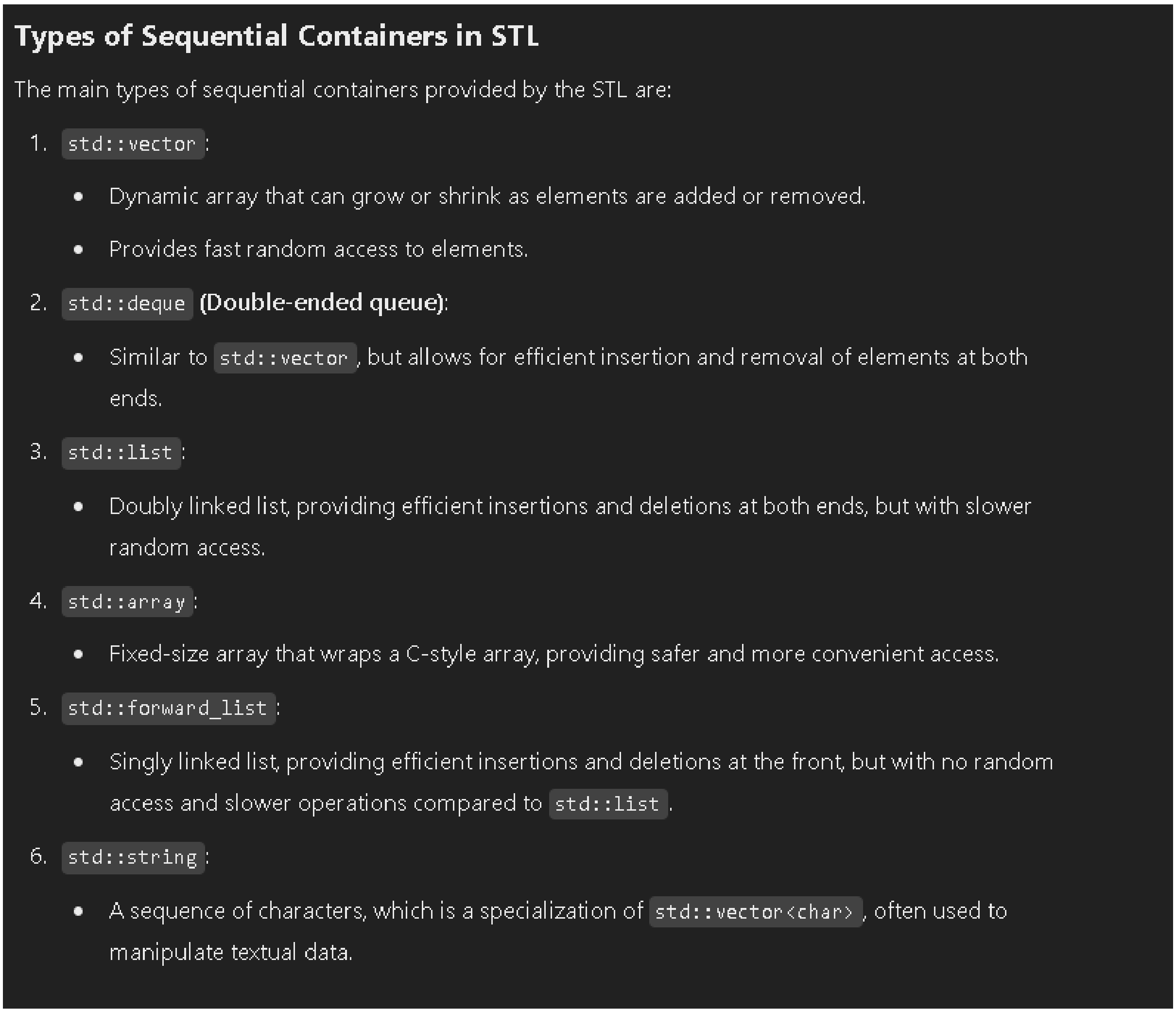
13) What is a sequential container? List various sequential containers. Compare arrays and vectors



(a) Sequence Containers

These containers store objects (i.e. data) in linear order. The new elements are added at the end of the existing collection of elements.

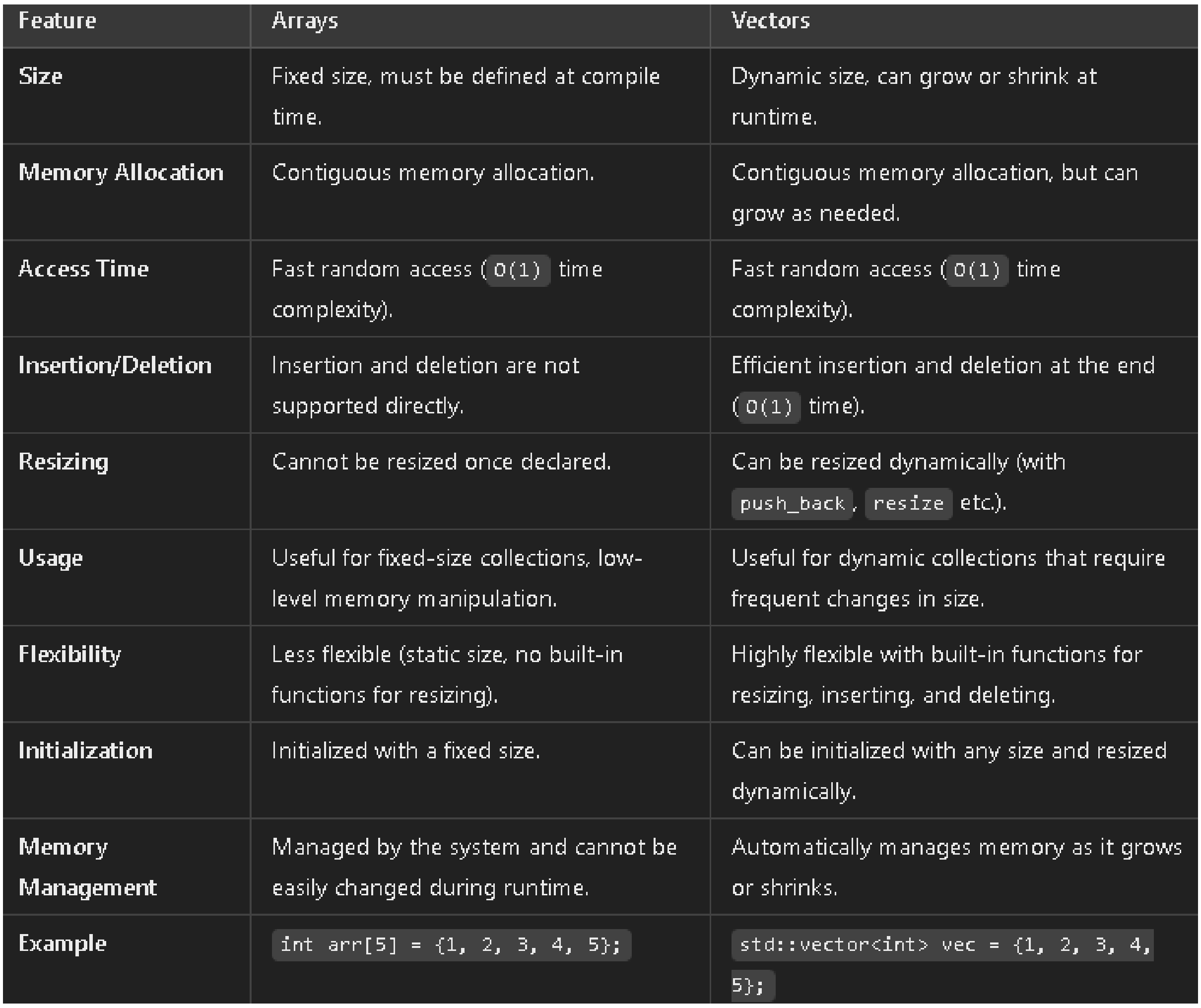
e.g. Vector, Array, List, Deque, forward_listareSequence STL Containers.



Types of Sequential Containers in STL

The main types of sequential containers provided by the STL are:

1. `std::vector`:
   - Dynamic array that can grow or shrink as elements are added or removed.
   - Provides fast random access to elements.
2. `std::deque` **(Double-ended queue)**:
   - Similar to `std::vector`, but allows for efficient insertion and removal of elements at both ends.
3. `std::list`:
   - Doubly linked list, providing efficient insertions and deletions at both ends, but with slower random access.
4. `std::array`:
   - Fixed-size array that wraps a C-style array, providing safer and more convenient access.
5. `std::forward_list`:
   - Singly linked list, providing efficient insertions and deletions at the front, but with no random access and slower operations compared to `std::list`.
6. `std::string`:
   - A sequence of characters, which is a specialization of `std::vector<char>`, often used to manipulate textual data.



Feature	Arrays	Vectors
Size	Fixed size, must be defined at compile time.	Dynamic size, can grow or shrink at runtime.
Memory Allocation	Contiguous memory allocation.	Contiguous memory allocation, but can grow as needed.
Access Time	Fast random access (`O(1)` time complexity).	Fast random access (`O(1)` time complexity).
Insertion/Deletion	Insertion and deletion are not supported directly.	Efficient insertion and deletion at the end (`O(1)` time).
Resizing	Cannot be resized once declared.	Can be resized dynamically (with `push_back`, `resize` etc.).
Usage	Useful for fixed-size collections, low-level memory manipulation.	Useful for dynamic collections that require frequent changes in size.
Flexibility	Less flexible (static size, no built-in functions for resizing).	Highly flexible with built-in functions for resizing, inserting, and deleting.
Initialization	Initialized with a fixed size.	Can be initialized with any size and resized dynamically.
Memory Management	Managed by the system and cannot be easily changed during runtime.	Automatically manages memory as it grows or shrinks.
Example	`int arr[5] = {1, 2, 3, 4, 5};`	`std::vector<int> vec = {1, 2, 3, 4, 5};`

COMPILED BY:

Karan Salunkhe, and Anish Joshi

Email: karan@pacificoracle.in

(16 Nov 2024 – 17 Nov 2024)