ASSIGNMENT-5

1. What are streams in C++ and why are they important?

Streams in C++ are abstract representations of input and output devices. They enable the transfer of data into and out of programs. A stream is essentially a sequence of bytes that acts as an interface between the program and input/output devices. The two main types of streams are input streams and output streams. Input streams are used to receive data (e.g., from the keyboard or files), while output streams are used to send data (e.g., to the screen or files). C++ includes the iostream library, which provides several stream classes such as `cin`, `cout`, `ifstream`, `ofstream`, etc. Streams help in handling data in a structured, easy, and consistent way. They also enable formatted and unformatted I/O operations. Streams abstract the complexities of device-level I/O, making programming easier and more readable.

2. Explain the different types of streams in C++.

C++ has several types of streams based on the direction and nature of data flow:

1. **Input Stream**: Used for input operations; examples include `cin`, `ifstream`.
2. **Output Stream**: Used for output operations; examples include `cout`, `ofstream`.
3. **I/O Stream**: Supports both input and output; e.g., `fstream`.
4. **String Stream**: Used to perform I/O on strings using `stringstream`.
   Streams can also be categorized into formatted and unformatted streams based on data representation. Formatted streams present data in a human-readable form, whereas unformatted streams deal with raw bytes. Each stream is associated with a buffer, which temporarily stores the data during transfer. These types help manage data flow between the program and various I/O devices.

3. How do input and output streams differ in C++?

Input and output streams in C++ differ primarily in the direction of data flow. **Input streams** bring data *into* the program from an external source, like a keyboard or file (`cin`, `ifstream`). **Output streams**, on the other hand, send data *out* of the program to an external destination, like the screen or a file (`cout`, `ofstream`). Input operations use the extraction operator (>>), while output operations use the insertion operator (<<). Input streams read characters or formatted/unformatted data, while output streams format and send data to display or storage. These streams are defined in <iostream> and <fstream> headers and play a vital role in I/O operations in C++ programs.

4. Describe the role of the iostream library in C++.

The `<iostream>` library in C++ provides essential functionalities for input and output operations. It defines the standard I/O stream objects like `cin`, `cout`, `cerr`, and `clog`. This library includes important stream classes such as `istream`, `ostream`, and `iostream`. It enables the use of stream operators (>>, <<) for formatted input and output. `<iostream>` abstracts low-level I/O operations and provides a user-friendly interface for console I/O. The library handles buffering and synchronization with C-style I/O automatically. It also supports both formatted and unformatted I/O operations. Overall, it is fundamental to all C++ programs that require interaction with the user or display of output.

5. What is the difference between a stream and a file stream?

A **stream** in C++ is a general interface for data flow (input/output) between the program and I/O devices. Common examples are `cin` and `cout`. A **file stream**, on the other hand, is a specialized type of stream that is associated with file handling. File streams use classes like `ifstream` for input, `ofstream` for output, and `fstream` for both. Unlike standard streams, file streams require file operations such as open, read/write, and close. File streams are defined in the `<fstream>` header and work with data stored in files on disk rather than immediate input/output devices like a console. Thus, file streams extend stream functionality for file-based data manipulation.

6. What is the purpose of the cin object in C++?

The `cin` object in C++ is an instance of the `istream` class used for standard input. It reads data from the keyboard and passes it to the program. `cin` works with the extraction operator (>>) to input formatted data like integers, floats, and strings. For example, `cin >> x;` reads a value into variable `x`. It handles whitespace and newline characters appropriately when reading input. `cin` supports chaining to read multiple values in a single line. It can also be used with `get()`, `getline()`, and `ignore()` functions for more control over input. It's essential for interactive console-based programs.

7. How does the cin object handle input operations?

The `cin` object handles input operations using the extraction operator (>>). It reads data from the standard input buffer and stores it in the provided variable. `cin` automatically skips whitespace (spaces, tabs, and newlines) before reading a value. For strings, it reads input up to

the first whitespace by default. To read full lines including spaces, functions like `getline(cin, str)` are used. It also supports functions like `cin.get()`, `cin.ignore()`, and `cin.peek()` to manage specific character-based input. If an input error occurs, `cin` sets error flags that can be checked using `cin.fail()` or cleared using `cin.clear()`.

8. What is the purpose of the cout object in C++?

The `cout` object in C++ is used for standard output. It is an instance of the `ostream` class and sends data to the console or display screen. The insertion operator (<<) is used with `cout` to display various data types including integers, floats, strings, and even user-defined types. For example, `cout << "Hello";` displays the text on the screen. `cout` supports manipulators like `endl`, `setw`, and `setprecision` to control the formatting of output. It's fundamental for interacting with users through the command line. `cout` makes output intuitive and readable, enhancing the user experience in console applications.

9. How does the cout object handle output operations?

The `cout` object handles output by using the insertion operator (<<) to send data from the program to the console. It converts variables into a readable string format and prints them on the screen. `cout` supports chaining, which allows multiple values to be printed in a single statement. It also works with manipulators such as `endl` to insert new lines, or `setw` to format width. Output using `cout` is buffered, meaning it temporarily stores data before displaying it to improve efficiency. The buffer is flushed when `endl` is used or when the program ends. `cout` ensures formatted, readable output for user interaction.

10. Explain the use of the insertion (<<) and extraction (>>) operators in conjunction with cin and cout.

The insertion (<<) and extraction (>>) operators are stream operators in C++. The **insertion operator (<<)** is used with `cout` to output data. For example, `cout << "Hello";` prints "Hello" to the screen. The **extraction operator (>>)** is used with `cin` to input data. For example, `cin >> x;` stores input into variable `x`. These operators are overloaded to support various data types. They can also be used with custom classes by overloading the operators. These operators make I/O operations concise and readable, improving code clarity. They enable chaining for multiple input/output expressions, simplifying complex I/O interactions.

11. What are file streams in C++ and how are they used?

File streams in C++ are used to perform input and output operations on files. They are part of the `<fstream>` header and include three main classes:

- `ifstream` (input file stream for reading),
- `ofstream` (output file stream for writing),
- `fstream` (for both reading and writing).
  To use a file stream, you must first declare an object of one of these classes and then associate it with a file using the `open()` function or constructor. For example:

```
cpp
CopyEdit
ifstream infile("data.txt");
ofstream outfile("result.txt");
```

After opening, you can use the >> and << operators or member functions like `read()`, `write()` to interact with the file. Finally, use `close()` to end the file connection. File streams enable persistent data storage and are crucial for real-world applications like saving user data or logs.

12. Differentiate between ifstream, ofstream, and fstream.

These are file stream classes defined in the `<fstream>` header:

- `ifstream` (input file stream): Used for reading from files. Example: `ifstream fin("data.txt");`
- `ofstream` (output file stream): Used for writing to files. Example: `ofstream fout("output.txt");`
- `fstream` (file stream): Used for both reading and writing. Example: `fstream file("file.txt", ios::in | ios::out);`

Differences:

- `ifstream` and `ofstream` are specialized classes with restricted functionality—input only or output only, respectively.
- `fstream` provides flexibility with modes like `ios::in`, `ios::out`, `ios::app`, `ios::binary`, etc.
- You can open, read, write, and close files using these classes.
  These distinctions help manage file I/O with better control and clarity in C++ programs.

13. How do you open and close a file in C++?

To open a file, use one of the file stream objects (`ifstream`, `ofstream`, or `fstream`) with the `open()` function or constructor:

```
cpp
CopyEdit
ifstream infile;
infile.open("input.txt");
```

Alternatively:

```
cpp
CopyEdit
ofstream outfile("output.txt");
```

To specify modes, use flags like `ios::in`, `ios::out`, `ios::app`, etc.:

```
cpp
CopyEdit
fstream file("data.txt", ios::in | ios::out);
```

After finishing operations, close the file using `close()`:

```
cpp
CopyEdit
infile.close();
outfile.close();
```

Always check if the file was successfully opened using `.is_open()` or by checking the stream's status:

```
cpp
CopyEdit
if (!infile.is_open()) {
    cout << "Error opening file";
}
```

Closing ensures data is written properly and releases system resources.

14. What are the different modes in which a file can be opened in C++?

In C++, files can be opened using various modes defined in the `ios` namespace:

1. `ios::in` – Opens file for reading.
2. `ios::out` – Opens file for writing. If file exists, it will be truncated.
3. `ios::app` – Opens file for appending data at the end.
4. `ios::ate` – Opens file and moves the cursor to the end.
5. `ios::trunc` – Truncates the content of file if it exists.
6. `ios::binary` – Opens file in binary mode.
   These modes can be combined using the bitwise OR operator (`|`):

```
cpp
CopyEdit
fstream file("data.txt", ios::in | ios::out | ios::app);
```

The correct mode must be chosen based on the operation to avoid data loss or read/write errors.

15. How do you read from a file using ifstream?

To read from a file using `ifstream`, follow these steps:

1. Include the `<fstream>` header.
2. Create an `ifstream` object.
3. Open the file using the constructor or `open()` method.
4. Use `>>`, `getline()`, or `get()` to read data.
5. Close the file using `close()`.

Example:

```
cpp
CopyEdit
ifstream file("input.txt");
string line;
while (getline(file, line)) {
    cout << line << endl;
}
file.close();
```

You can also read character-by-character using `file.get(ch)` or word-by-word using `file >> word`. Always check `file.is_open()` before reading and use `eof()` to detect the end of the file.

16. How do you write to a file using ofstream?

To write to a file using `ofstream`:

1. Include the `<fstream>` header.
2. Declare an `ofstream` object.
3. Open a file using the constructor or `open()`.
4. Use the `<<` operator to write data.
5. Close the file using `close()`.

Example:

```
cpp
CopyEdit
ofstream file("output.txt");
file << "Hello, File!" << endl;
file.close();
```

You can also use `write()` for binary or formatted output:

```
cpp
CopyEdit
file.write(reinterpret_cast<char*>(&data), sizeof(data));
```

Writing overwrites existing content unless opened in `ios::app` mode. Always check if the file opens successfully before writing.

17. What is the significance of file pointers in C++?

File pointers are internal markers that indicate the current position in a file. There are two types:

1. **get pointer (input position pointer)**: Used for reading.
2. **put pointer (output position pointer)**: Used for writing.

You can manipulate these pointers using:

- `seekg()`, `tellg()` for input streams.
- `seekp()`, `tellp()` for output streams.

Example:

```
cpp
CopyEdit
file.seekg(0, ios::end);
```

```
int size = file.tellg();
```

This mechanism allows random access, enabling reading/writing at specific file locations. File pointers are essential for binary file operations and modifying large files efficiently.

18. How do you use seekg() and seekp() functions in C++?

`seekg()` and `seekp()` are used to move the **get** and **put** file pointers, respectively:

- `seekg(offset, direction)` moves the input pointer.
- `seekp(offset, direction)` moves the output pointer.
  Directions can be:
- `ios::beg` (beginning)
- `ios::cur` (current position)
- `ios::end` (end of file)

Example:

```
cpp
CopyEdit
ifstream fin("data.txt");
fin.seekg(10, ios::beg); // Moves get pointer to 10th byte from
start

ofstream fout("data.txt");
fout.seekp(-5, ios::end); // Moves put pointer 5 bytes before
end
```

These functions allow random access, important for non-linear file operations like updating specific records.

19. What are the advantages of using file handling in C++?

File handling provides several benefits:

1. **Persistent Storage**: Data remains after program execution ends.
2. **Large Data Management**: Useful for handling large datasets.
3. **Input/Output Flexibility**: Enables both reading and writing.
4. **Random Access**: Allows accessing specific locations in a file using pointers.
5. **Data Sharing**: Files can be shared across different systems or applications.
6. **Data Logging**: Ideal for saving logs, configurations, or user data.
7. **Security**: Data can be encrypted/stored securely.

8. **Automation**: Enables automated data processing and report generation.
9. **Backup**: Files can serve as backups for critical information.
10. **User Interaction**: Programs can save and load user settings or progress.

20. How can you check if a file is successfully opened in C++?

You can check if a file was successfully opened using:

1. `.is_open()` method:

```
cpp
CopyEdit
ifstream file("data.txt");
if (!file.is_open()) {
    cout << "Failed to open file" << endl;
}
```

2. Checking the stream's status directly:

```
cpp
CopyEdit
if (file) {
    // File is open and ready
} else {
    // Error occurred
}
```

3. For failure detection:

```
cpp
CopyEdit
if (file.fail()) {
    cout << "Open operation failed";
}
```

Always perform such checks before reading/writing to avoid runtime errors or data corruption.

21. What is the use of tellg() and tellp() in file handling?

In C++, `tellg()` and `tellp()` are used to **get the current position of file pointers**:

- `tellg()` returns the position of the get (input) pointer.
- `tellp()` returns the position of the put (output) pointer.

These functions return a value of type `streampos` representing the byte offset from the beginning of the file.

Example:

```
cpp
CopyEdit
ifstream fin("example.txt");
fin.seekg(0, ios::end);
cout << "File size: " << fin.tellg(); // Displays file size in
bytes
```

They are commonly used for:

- Determining the size of a file.
- Keeping track of where to read or write next.
- Implementing file operations like resume or partial downloads.
  They enhance random access capability in file processing.

22. Explain the concept of binary files in C++.

Binary files store data in the same format as in memory (i.e., raw bytes), unlike text files that store data as readable characters.

Characteristics:

- Faster read/write.
- Compact size.
- Not human-readable.
- Suitable for storing structures, images, executables.

To use:

1. Open file in `ios::binary` mode:

```
cpp
CopyEdit
ofstream fout("data.dat", ios::binary);
```

2. Use `write()` and `read()` functions:

```
cpp
CopyEdit
fout.write((char*)&obj, sizeof(obj));
fin.read((char*)&obj, sizeof(obj));
```

Advantage: Saves entire object in one go.
Caution: Only works reliably with POD (Plain Old Data) types.

23. How do you read and write binary files in C++?

Use `read()` and `write()` methods from file streams:

- `read(char* buffer, int size)` – reads raw bytes into memory.
- `write(const char* buffer, int size)` – writes raw bytes from memory.

Example (writing):

```
cpp
CopyEdit
ofstream fout("data.dat", ios::binary);
fout.write((char*)&obj, sizeof(obj));
fout.close();
```

Example (reading):

```
cpp
CopyEdit
ifstream fin("data.dat", ios::binary);
fin.read((char*)&obj, sizeof(obj));
fin.close();
```

Steps:

1. Open the file with `ios::binary`.
2. Cast object address to `char*`.
3. Use correct size to prevent corruption.

Binary file I/O is essential for high-performance applications and storing structured data efficiently.

24. What is the difference between text and binary files?

Feature	Text File	Binary File
Format	Human-readable characters	Raw bytes
Size	Larger (includes formatting)	Smaller (no formatting)
Speed	Slower (char-by-char processing)	Faster (block I/O)
Usage	Logs, configs, readable data	Images, executables, structured
Editing	Can be edited with text editors	Needs specialized software
Functions Used	<<, >>, `getline()`	`read()`, `write()`

Use text files for readability and debugging, binary files for performance and compact storage.

25. How do you handle errors in file operations?

C++ provides multiple ways to detect and handle file errors:

1. **Check if file opened:**

```
cpp
CopyEdit
ifstream fin("data.txt");
if (!fin) {
    cout << "Error opening file.";
}
```

2. **Check for I/O failure:**

- `.fail()` – logical error (e.g., wrong data type).
- `.bad()` – read/write failure.
- `.eof()` – end-of-file reached.

3. **Clear flags:**

```
cpp
CopyEdit
fin.clear();  // Resets error flags
```

4. **Exception handling (optional):**

```
cpp
CopyEdit
fin.exceptions(ifstream::failbit | ifstream::badbit);
try {
    fin.open("file.txt");
} catch (...) {
```

```
    cout << "Exception during file operations.";
}
```

Always ensure files are closed properly using `.close()` to avoid data loss.

26. What is the purpose of ios::in, ios::out, and ios::app?

These are file mode flags used when opening files:

- **`ios::in`** – Open file for reading:

```
cpp
CopyEdit
ifstream fin("file.txt", ios::in);
```

- **`ios::out`** – Open file for writing. Truncates if file exists:

```
cpp
CopyEdit
ofstream fout("file.txt", ios::out);
```

- **`ios::app`** – Opens file in append mode. Writes go to end of file:

```
cpp
CopyEdit
ofstream fout("log.txt", ios::app);
```

These flags can be combined using the OR operator (`|`):

```
cpp
CopyEdit
fstream file("example.txt", ios::in | ios::out | ios::app);
```

These modes give control over how data is processed and saved.

27. Explain the use of eof() function in file handling.

`eof()` stands for **End Of File**. It returns `true` when the file pointer reaches the end of a file.

Use case:

```
cpp
```

```
CopyEdit
ifstream file("data.txt");
string line;
while (!file.eof()) {
    getline(file, line);
    cout << line << endl;
}
```

Better way (recommended):

```
cpp
CopyEdit
while (getline(file, line)) {
    cout << line << endl;
}
```

`eof()` is best used after a failed read to confirm it's due to end-of-file. It helps avoid reading garbage values and infinite loops.

28. How can you copy content from one file to another in C++?

Steps:

1. Open source file in `ifstream`.
2. Open destination file in `ofstream`.
3. Read from source and write to destination.

Example:

```
cpp
CopyEdit
ifstream fin("source.txt");
ofstream fout("dest.txt");
string line;
while (getline(fin, line)) {
    fout << line << endl;
}
fin.close();
fout.close();
```

You can also copy in binary:

```
cpp
CopyEdit
```

```
char ch;
while (fin.get(ch)) {
    fout.put(ch);
}
```

This is useful in file backup, file transformation, and data migration programs.

29. What is serialization in C++ file handling?

Serialization is the process of converting an object's state into a format that can be stored (file) or transmitted (network), then later reconstructed.

In C++, manual serialization is required, especially for complex objects:

```
cpp
CopyEdit
fout.write((char*)&obj, sizeof(obj)); // Serialization
```

Deserialization:

```
cpp
CopyEdit
fin.read((char*)&obj, sizeof(obj));
```

Serialization allows:

- Saving objects to disk.
- Transmitting objects over networks.
- Restoring previous states.

More advanced serialization uses libraries like Boost or Cereal.

30. How do you update a record in a file in C++?

To update a record:

1. Open file in `ios::in | ios::out | ios::binary`.
2. Use `seekp()` to go to the correct position.
3. Overwrite the record using `write()`.

Example:

```
cpp
CopyEdit
fstream file("data.dat", ios::in | ios::out | ios::binary);
file.seekp(recordIndex * sizeof(obj), ios::beg);
file.write((char*)&updatedObj, sizeof(updatedObj));
```

This works well for fixed-size records (like structs).
For variable-length records, consider rewriting the whole file using a temporary file.

31. What is a template in C++?

A **template** is a feature in C++ that allows writing **generic and reusable code**. It enables functions or classes to work with different data types without rewriting code for each type.

Syntax:

```
cpp
CopyEdit
template <class T>
T add(T a, T b) {
    return a + b;
}
```

You can call it with different types:

```
cpp
CopyEdit
cout << add<int>(3, 4);         // 7
cout << add<float>(3.1, 4.2);  // 7.3
```

Templates reduce code duplication and improve maintainability. They are the foundation of STL (Standard Template Library).

There are **function templates** and **class templates**. Templates help write type-safe and efficient programs.

32. Differentiate between function template and class template.

Feature	Function Template	Class Template
Purpose	Generalizes a single function	Generalizes an entire class
Syntax	`template <class T> T func(T a)`	`template <class T> class ClassName {}`

Feature	Function Template	Class Template
Use Case	Generic operations like add, max, swap	Generic data structures like Stack, Queue
Example	`add<int>(a, b)`	`Stack<int> s;`
Scope	Only function logic	Class members, data, and methods

Function templates are best for operations. Class templates are best for reusable structures.

33. Write a program to demonstrate function template.

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

template <class T>
T getMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max of 4 and 7: " << getMax(4, 7) << endl;
    cout << "Max of 3.3 and 2.2: " << getMax(3.3, 2.2) << endl;
    cout << "Max of 'a' and 'z': " << getMax('a', 'z') << endl;
    return 0;
}
```

This function compares any two values and returns the greater one, regardless of data type. It demonstrates how templates improve code flexibility.

34. Write a program to demonstrate class template.

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

template <class T>
class Calculator {
    T a, b;
public:
    Calculator(T x, T y) : a(x), b(y) {}
```

```
    void add() { cout << "Sum: " << a + b << endl; }
    void multiply() { cout << "Product: " << a * b << endl; }
};

int main() {
    Calculator<int> c1(3, 4);
    c1.add();
    c1.multiply();

    Calculator<float> c2(2.5, 3.5);
    c2.add();
    c2.multiply();
    return 0;
}
```

This example shows how a class template can work with different data types without rewriting code.

35. What is STL in C++?

STL stands for **Standard Template Library**. It provides:

- **Containers**: Dynamic structures (vector, list, map).
- **Algorithms**: Sorting, searching, etc.
- **Iterators**: Navigate through containers.

Advantages:

- Generic, reusable code.
- Faster development.
- Type safety and efficiency.

Example:

```
cpp
CopyEdit
vector<int> v = {1, 2, 3};
sort(v.begin(), v.end());
```

STL is built on templates and offers high-performance, flexible components for managing collections of data.

36. Differentiate between vector and list in STL.

Feature	Vector	List
Structure	Dynamic array	Doubly linked list
Access	Fast random access (indexing)	No random access
Insertion	Slow at beginning/middle	Fast at any position
Memory	Contiguous memory	Non-contiguous memory
Header	`<vector>`	`<list>`

Use `vector` when you need fast access, `list` when you frequently insert/delete in the middle.

37. Explain the use of map in STL.

`map` is an **associative container** in STL that stores elements in key-value pairs, with **unique keys** and sorted order.

Syntax:

```
cpp
CopyEdit
map<string, int> marks;
marks["Alice"] = 90;
marks["Bob"] = 85;
```

Features:

- Fast retrieval via key.
- Ordered by keys.
- Supports iterators.

Example:

```
cpp
CopyEdit
for (auto& p : marks) {
    cout << p.first << " got " << p.second << " marks\n";
}
```

Ideal for lookup tables, dictionaries, and indexing.

38. How can you sort a vector using STL?

Use `sort()` from `<algorithm>` to sort a vector.

Example:

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

int main() {
    vector<int> v = {5, 2, 9, 1};
    sort(v.begin(), v.end()); // ascending
    for (int x : v) cout << x << " ";

    // descending
    sort(v.begin(), v.end(), greater<int>());
    return 0;
}
```

`sort()` works with iterators, allowing custom comparators for complex sorting.

39. What are iterators in STL?

Iterators are **pointers-like objects** used to traverse STL containers.

Types:

- `begin()` – points to first element.
- `end()` – points past the last element.

Example:

```
cpp
CopyEdit
vector<int> v = {10, 20, 30};
for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << " ";
}
```

Iterators provide a uniform interface for accessing elements, regardless of container type. They enable STL algorithms to work seamlessly.

40. Write a program to demonstrate the use of vector and iterator.

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

int main() {
    vector<int> nums = {10, 20, 30, 40, 50};

    cout << "Using iterator:\n";
    vector<int>::iterator it;
    for (it = nums.begin(); it != nums.end(); ++it) {
        cout << *it << " ";
    }
    return 0;
}
```

This program uses a vector to store integers and an iterator to display them. It demonstrates how iterators work with vectors in STL.

41. What is exception handling in C++?

Exception handling in C++ provides a mechanism to detect and handle **runtime errors** in a structured and safe way using `try`, `catch`, and `throw`.

Syntax:

```
cpp
CopyEdit
try {
    // code that might throw
    if (x == 0) throw "Divide by zero!";
} catch (const char* msg) {
    cout << "Error: " << msg;
}
```

Keywords:

- `try`: Defines block with risky code.
- `throw`: Throws an exception.
- `catch`: Catches and handles the exception.

It prevents program crashes and ensures **graceful failure**. C++ also supports custom exception types using classes.

42. Write a program to demonstrate exception handling.

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

int divide(int a, int b) {
    if (b == 0)
        throw "Division by zero error!";
    return a / b;
}

int main() {
    int x = 10, y = 0;
    try {
        cout << "Result: " << divide(x, y);
    } catch (const char* e) {
        cout << "Exception caught: " << e;
    }
    return 0;
}
```

This program catches and handles a divide-by-zero error using exception handling in C++.

43. What is the purpose of try, catch, and throw in exception handling?

These three keywords form the backbone of exception handling in C++:

- **`try` block**: Contains code that might cause an exception.
- **`throw` keyword**: Used to signal (raise) an exception.
- **`catch` block**: Handles the exception thrown.

Example:

```
cpp
CopyEdit
try {
    throw 404;
```

```
} catch (int e) {
    cout << "Error code: " << e;
}
```

This structure ensures **error detection, isolation, and safe resolution**. It avoids crashes and allows developers to deal with unexpected conditions.

44. What is a generic exception handler?

A **generic exception handler** catches all types of exceptions using the `catch(...)` syntax.

Example:

```
cpp
CopyEdit
try {
    throw 3.14;
} catch (...) {
    cout << "Caught some exception!";
}
```

Use this when:

- You don’t know the exact type of exception.
- You want to handle any error uniformly.

It's a **catch-all fallback** but provides **no type information**, so it should be used after more specific `catch` blocks.

45. Can we handle multiple exceptions in a program?

Yes, C++ allows **multiple catch blocks** for handling different types of exceptions.

Example:

```
cpp
CopyEdit
try {
    throw 'x';
} catch (int e) {
    cout << "Integer exception";
} catch (char e) {
```

```
    cout << "Character exception";
} catch (...) {
    cout << "Generic exception";
}
```

Each `catch` handles a specific data type. Order matters – most specific should come first.

This allows fine-grained control and specific recovery for different error conditions.

46. What is stack unwinding in exception handling?

Stack unwinding is the process where the **compiler destroys all local objects** in reverse order of construction when an exception is thrown and not caught in the same scope.

Example:

```
cpp
CopyEdit
class Test {
public:
    Test() { cout << "Constructor\n"; }
    ~Test() { cout << "Destructor\n"; }
};

void func() {
    Test t;
    throw 10;
}
```

When `throw` is called, the destructor of `Test` will be called before transferring control to `catch`. This automatic cleanup ensures resource safety during exception propagation.

47. What are the advantages of exception handling?

1. Improves **program reliability** by preventing crashes.
2. Separates **error handling** from normal code.
3. Supports **hierarchical error handling** via multiple catch blocks.
4. Enables **custom exceptions** for domain-specific errors.
5. Ensures **resource cleanup** via destructors (RAII).
6. Allows use of `catch(...)` for unknown errors.
7. Makes code **cleaner** and more readable.
8. Prevents the use of error-prone return codes.

9. Supports both **standard and user-defined exceptions**.
10. Ensures **modular exception handling** across functions.

48. What is a class template?

A **class template** allows creating **generic classes** that can operate with different data types.

Syntax:

```
cpp
CopyEdit
template <class T>
class MyClass {
    T data;
public:
    MyClass(T d) : data(d) {}
    void display() { cout << data; }
};
```

Usage:

```
cpp
CopyEdit
MyClass<int> obj1(10);
MyClass<string> obj2("Hello");
```

This reduces duplication, supports reusability, and forms the basis of many STL containers.

49. Write a program to demonstrate class template with multiple parameters.

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

template <class T1, class T2>
class Pair {
    T1 a;
    T2 b;
public:
    Pair(T1 x, T2 y) : a(x), b(y) {}
    void show() {
        cout << "First: " << a << ", Second: " << b << endl;
```

```
    }
};

int main() {
    Pair<int, float> p1(10, 3.5);
    Pair<string, int> p2("Age", 21);
    p1.show();
    p2.show();
    return 0;
}
```

This class template accepts multiple types, demonstrating flexibility and code reusability in C++.